ASYNCHRONOUS DESIGN FOR UBIQUITOUS COMPUTING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Carlos Tadeo Ortega Otero

August 2014

ASYNCHRONOUS DESIGN FOR UBIQUITOUS COMPUTING

Carlos Tadeo Ortega Otero, Ph.D.

Cornell University 2014

Computing in the 21^{st} century is rapidly moving from the personal computer model into area of ubiquitous computing, in which technology will be woven into the fabric of everyday life. This ubiquitous, pervasive computing is enabled by Wireless Sensor Networks (WSNs). Engineers deploy these WSNs in myriad applications, ranging from implanted medical monitoring devices to industrial control systems.

Node energy—which translates to lifetime— and throughput are crucial metrics for evaluating wireless sensor nodes. Previously, energy reduction at the cost of performance was a common engineering trade-off for mote microcontrollers. However, increased application complexity requires greater computational power while retaining a low-power envelope. The reduced energy consumption and increased processing power enables more complex operations on collected data to be performed locally, reducing the use of energy-hungry wireless transmission systems and in turn the overall energy consumption.

In this thesis, we study the use of Quasi Delay-Insensitive (QDI) circuits to design a microcontroller that fits the WSN application space. All of our architectural and circuit decisions aim to maximize the sensor node lifetime and while retaining performance in a WSN system.

Security of WSN data is also critical, especially in medical or federal use cases. To address this need, we present the analysis and design of software, hardware, and hybrid AES implementations. Again, we optimized for high encryption performance while maintaining node lifetime.

BIOGRAPHICAL SKETCH

Carlos Tadeo Ortega Otero was born in Ciudad de México, México where he grew up. Carlos is the son of Rodolfo Ortega Mendoza, and Myrna Otero Berdon. His dad is a retired Electrical Engineer that worked for more than 35 years at the Mexican state-owned petroleum company: PEMEX. Carlos' mom is a homemaker. Carlos completed his primary and high school education at the Instituto Juventud del Estado de México.

He received his first computer as a gift in the summer of 1992. Since then, he became passionate about the software and hardware of computer systems. The summer after the end of elementary school, he enrolled in a program to learn Basic, Visual Basic, databases and basic command line scripting using MS-DOS batch files. In the year 2000, he bought and installed Corel Linux. He graduated high school with honors in 2001.

He began his undergraduate studies in Computer Engineering in 2006. Carlos' first contact with research was an invitation from Prof. Marcos de Alba Rosano to augment a cycle-accurate architectural simulator to give detailed power estimations of each pipeline stage of a superscalar Alpha microprocessor.

In his senior year of college, Carlos was admitted to the Cornell's International Undergraduate Engineering Research Internship/Mexico (IUERIM), organized by Prof. Francisco Valero-Cuevas. During the IUERIM program, he worked at the Computer Systems Laboratory under the tutelage of Prof. Rajit Manohar. The result of this internship was a technical report that described automatic mechanisms to adapt the slack of control signals on a split-merge datapath.

He returned to Mexico to conclude his undergraduate studies, receiving in 2006 his B.S. degree in Computer Engineering from the Instituto Tecnológico de Monterrey, campus Estado de México.

After concluding his bachelors, he worked at Softtek, a leader in software development accorss the world. His main interest was to create interfaces to safely interact with point of sale devices.

Carlos started his graduate studies at Cornell University in 2006 as a Cornell-CONACyT fellow, under the tutelage and advise of Prof. Rajit Manohar. As a graduate student, he collaborated in several Computer Architecture and VLSI projects. During his stay at Cornell, he co-authored 9 papers in peer-review conferences, and was a core team member in the design, implementation and testing of 8 test chips. In 2012, Carlos interned at IBM as part of the Synapse team led by Dharmendra S. Modha, where he entered the fascinating world of neuromorphic computing.

He received his Master of Science degree from Cornell University in 2012. The title of his M.S. thesis is: Static Power Reduction Techniques for Asynchronous Circuits. His research interests include low-energy Very Large Scale Integration (VLSI), computer-aided design tools for VLSI synthesis and implementation, asynchronous integrated circuit design, emerging VLSI technologies, secure VLSI systems, wireless sensor network, home automation, and sensor network nodes in ubiquitous computing.

Carlos is part of the Asynchronous VLSI Group and Architecture (AVLSI) led by Professor Rajit Manohar. After graduation, Carlos will continue as a post-doctoral scholar. Carlos wishes to stay in academia with strong relationships and interaction with industrial development design teams and laboratories.

When not doing asynchronous circuits, Carlos may be found having fun with his wife and daughter, practicing some sport, or enjoying an evening conversation with a friend.

A teacher affects eternity;

he can never tell where his influence stops.

Henry B. Adams

This thesis is dedicated to all my educators, teachers, and professors

ACKNOWLEDGEMENTS

This research project was only possible with the kind support from many people. Undoubtedly, the most influential person in my career is Prof. Rajit Manohar. Rajit's passion for VLSI and Asynchronous circuits has encouraged me to learn in depth different concepts and link them together to build new ideas. His unsurpassed knowledge of various areas of VLSI and Computer Science has been extremely resourceful. My interaction with Rajit is better described by a Chinese proverb that says: *"A single conversation with a wise man is better than ten years of study"*. I would also like to thank the faculty at the Computer Systems Laboratory for their thoughtful comments throughout my graduate studies.

I thank the members of my committee: Prof. Andrew C. Myers, Edward G. Suh and Prof. Rajit Manohar.

I thank Prof. Amit Lal and Prof. Sunil Bhave for allowing me to work with them on interesting projects that push the limits of Asynchronous Circuits.

Prof. Marcos de Alba Rosano was the first person that introduced me to research in electronics and computer architecture. He invited me to work on a research paper. I am really thankful for that, since that experience was a sway that started me as an incipient researcher. I thank Prof. Francisco Valero Cuevas for investing time in me, and helping me to get an undergraduate research opportunity that paved the way for my graduate studies.

David Fang and Filipp Akopyan were my first mentors at Cornell. They were the first who taught me Asynchronous Circuits, VLSI, and how to use the different tools available in the lab. Filipp has always encouraged me to work harder each day to achieve my goals. Ilya Ganusov and Basit Riaz always made research seem fun and fascinating. Nabil Imam and Stephen Longfield taught me how to keep focused on research and how make impact with it. I also thank the new(er) members of the AVLSI group: Julia, Ned, Nicholas, and Tayyar for reminding

me how to be passionate about graduate school. Thank you Kyle† for your collegiality, professionalism, and friendship; your invisible presence was felt during the composition of this thesis. Last, but not least, I particularly want to thank the λ-team: Benjamin Hill, Robert Karmazin and Jonathan Tse. They have been my closest research collaborators and friends throughout graduate school. I will always remember those sleepless nights in the laboratory weeks before a tapeout (and we had many of those). I am very thankful for the insightful discussions with the λ-team. I learned a lot on each one of them. Jon, thanks for all your help and for reading this thesis[1].

I would like to thank the ECE staff. In particular, I would like to thank Scott Coldren and Daniel Ritcher. Without them, ECE department would be a debacle.

My family has been a veritable network of support through difficult times during my research. My wife, Bistra Dilkina, has always been next to me, and every day creates a better version of me. Bistra has seen me through the best and the worst of graduate school. I thank her for share every day next to me. My daughter, Natalia, always gives me the motivation I need to work harder. I thank my parents Myrna and Rodolfo for their unconditional love, support, and for all the lessons they have taught me – *"Mamá, papá. ¡Gracias, los amo!"*. I would also like to thank my sister Myrna and my brother Rodolfo for always being a special part of my life. Thank you Edgar and the Velázquez-Armendáriz family for motivating to apply school and help me through difficult personal times. Isauro Salvador, Lourdes, Mario, and Mauricio thank you for always being next to me.

Finally, I would like to thank the funding agencies for their generous support and contribution to the projects I have worked on. I have always put all my effort that all the money I received to support my studies translates in interesting, useful and trustworthy research.

[1] You're welcome! -Jon

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS

LIST OF ABBREVIATIONS

ACT	Asynchronous Circuit Toolkit
AES	Advanced Encryption Standard
AVLSI	Asynchronous VLSI
BGMOS	Boosted-Gate CMOS
BSIM	Berkeley Short-channel IGFET Model
BOM	Bill of Materials
CAD	Computer-Aided Design/Development
CHP	Communicating Hardware Processes
CO	Cut-Off power gating
(C/N/P)CMOS	(Complementary/*n-type*/*p-type*) Metal Oxide Semiconductor
CONACyT	Consejo Nacional de Ciencia y Tecnologia
DRAM	Dynamic Random Access Memory
DRC	Design Rule Checker
EDA	Electronic Design Automation
FIPS	Federal Information Processing Standards
GIDL	Gate-Induced Drain Leakage
HSE	Handshaking Expansion
(H/S/L)VT	(High/Standard/Low)- Voltage Threshold transistor
ISA	Instruction Set Architecture
IP	Intellectual Property
gvddv	gated Vdd
gvssv	gated Vss
NTV	Near Threshold Voltage
PDN	Pulldown transistor network
PRS	Production Rule Set, digital pullup and pulldown description
PUP	Pullup transistor network
QDI	Quasi Delay-Insensitive
SPICE	Simulation Program with Integrated Circuit Emphasis
SRAM	Static Random Access Memory
ZDRTO	Zero-Delay Ripple Turn On
V_{DD}	Common drain voltage
VLSI	Very Large Scale Integration
Vss	Common source voltage
WSN	Wireless Sensor Network

CHAPTER 1

INTRODUCTION

Computing is not about computers any more.

It is about living

Nicholas Negroponte

Our world is becoming increasingly connected and instrumented with sensors, actuators, and data processors. This grid of smart elements will rapidly shift the 21^{st} century computing paradigm from the personal computer model into a configuration where technology "weaves into the fabric of everyday life" [70]. A myriad of applications are enabled by this *ubiquitous* computing, ranging from biological microorganism detection, implanted medical monitoring systems, battlefield surveillance, to industrial sensing devices [1,70]. The day when intelligent systems intertwine seamlessly with everyday life is inevitably happening, and it is arriving quickly.

Ubiquitous or *pervasive* computing is enabled by Wireless Sensor Networks (WSN). WSNs are networks consisting of fine-grained, spatially distributed nodes that *gather* data from our environment through sensors and can *process*, *transmit*, and *act* or *react* to the sensed stimuli [1].

WSNs comprise many small, low-cost nodes or *motes* that gather, process and propagate data about their surrounding environment. Deployment lifetimes can exceed several months, making node lifetime a crucial metric in the design space. However, improved node lifetime should not come at the cost of throughput. Increasing application complexity requires more processing execution power, forcing more aggressive peak performance targets for sensor nodes. Part of this is driven

by the ever growing complexity of application requirements, and part of it is driven by the high cost of wireless communication—doing more computation at the sensing node and transmitting less information is better operating point from a system lifetime perspective than transmitting raw data over the wireless link [1].

Off-the-shelf solutions for microcontrollers used in WSNs include general purpose processors, and embedded microcontrollers. On one side, general purpose processors include Intel's Quark, and ARM processors. On the other hand, embedded microcontrollers include the Texas Instruments MSP430 [25], Renesas RL78 [61], Silicon Labs Energy Micro, Microchip "nanowatt" PICs [44], and Atmel AVR "Pico Power" based processors [4,5].

General purpose processors are more powerful than their embedded counterparts. For example, the Quark SoC X1000 processor can run at 400 MHz and provides high speed PCI Express, and Ethernet interfaces. However Quark's 2 W power consumption makes its use prohibitive for applications where the desired deployment time is several months or even years.

While embedded processors provide multiple advantages over general-purpose microcontrollers, they do not necessarily fit the WSN paradigm. This potential mismatch arises partly because of low-performance rating, and partly because the difficulty in handling the event-driven nature of WSN computation.

Table 1.1 shows the energy consumption and performance of readily available off-the-shelf microcontrollers. At the time of this writing, none of them offer performance in excess of 50 MHz. As for power at 50 MHz, Atmel's ARM SAM4L processor uses 28 mW of power. A common 35 mA h, 3 V CR1220 battery will be fully depleted within 4 hours of constant operation. This example highlights the

Table 1.1: Energy consumption of commodity processors used in WSNs

Processor	Energy µA/MHz			Voltage V
Atmel ARM SAM4L [3]	90	@	12 MHz	3.3
	212	@	12 MHz	1.68
	180	@	48 MHz	3.3
AVR ATtiny13A [4]	190	@	1 MHz	1.8
	450	@	8 MHz	5.0
PIC 10LF320 [44]	35	@	0.5 MHz	1.8
	37	@	16 MHz	2.0
Renesas RL78 [61]	190	@	10 MHz	3.0
	154	@	24 MHz	3.0
TI MSP430-CC430 [25]	160	@	8 MHz	1.8
	225	@	20 MHz	2.4
TI MSP430-L092 [26]	31	@	0.125 MHz	0.9
	45	@	5 MHz	1.3

need for an energy efficient and powerful processor that can improve the lifetime of a deployed node.

Fortunately, most sensor network applications are bursty e.g. executing only when sensor data is available then returning to a quiescent state. This quiescent or sleep state could be significantly longer than the execution period, so minimizing power consumption during this idle phase is of paramount importance.

While in active mode, research shows that aggressive energy reduction at the cost of performance is a common trade-off for WSN microcontrollers. Some examples in this space include the MICA2 [23] and Smartdust [58] motes. MICA2 uses commercially available Atmel 128L microcontrollers, which provide good performance but relatively high energy consumption. In contrast, the Smartdust microcontroller is a single-pipeline RISC microcontroller that uses only 12 pJ per instruction but runs at a frequency of 500 kHz. At the extreme of the tradeoff space is the Phoenix microcontroller, which uses only 2.8 pJ per instruction but

runs at only 106 kHz [65].

While energy reduction in embedded microcontrollers is of paramount importance, it should not at a high cost to the performance. In addition, energy expenditure in data processing is much less compared to the energy cost of data communication. A high-performance microcontroller can potentially reduce the total energy consumed by enabling more data processing or compression on the local node, thus limiting expensive transmission on the raw data link [1,31,59]. Pottie, Kaiser et al. [59], show an example of the effects of fast local data processing. They demonstrate that in a multi-hop WSN topology, the energy cost of transmitting 1KB over a distance of 100m through a multi-hop network configuration, is approximately equivalent the same of executing 3 million instructions on a 100MIPS/W processor.

A fast microcontroller can potentially enable WSN global optimizations. By compressing and reducing the transmitted data size, for example, less bandwidth is required for sending and receiving data. A slow microcontroller would not be able to perform such tasks without incurring performance lag and increased response time, which might be critical for real-time applications. Compression and fast local data processing are effective mechanisms to utilize the limited resources of a WSN [1,59].

From the application standpoint, the system should consume minimal energy but deliver enough throughput to run critical applications such as an in situ health monitoring system. Examples of an ambulatory in situ monitoring system might include Electrocardiography (ECG), Electroencephalography (EEG), or Electromyography (EMG). Recent work on embedded systems has reported microcontrollers with energy consumption of only a few pJ/cycle [45], but their low

throughput only allows them to perform basic ECG algorithms. As system capable of throughput ranging from 50 MHz to 100 MHz has sufficient computational power to run complex ECG, EMG and EEG algorithms. This provides attractive features such as adaptive noise reduction, motion artifact cancellation, or feature extraction [2,13,63]. As a result of these applications as well as others just now emerging, much of the current research in ubiquitous systems has been focused on developing low voltage/energy microcontrollers with more computational power [2,22].

In this thesis, we present the design and implementation details of an Ultra-Low power Sensor Network Asynchronous Processor (ULSNAP) that is targeted at the sensor-node application space. Its advanced circuit design and event driven architecture provide increased horsepower with a minimal increase in energy consumption. The *measured* numbers of our ULSNAP test chip show that when idle, our chip consumes only 9 µW, with leakage power as the only contributor. It also has a fast wake-up time, transitioning from idle to active in only 6.5 ns. When active, our 90 nm chip delivers 93MIPS at 1.2 V and 47 MIPS at 0.95 V using 47 pJ and 29 pJ per cycle, respectively. ULSNAP is Pareto-optimal in the energy-performance space relative to other state of the art microcontrollers in its class, delivering more performance and increased node lifetime compared to readily available microcontrollers.

We achieved these results by exploiting the bursty operation of WSN to improve node lifetime. By paring ULSNAP with static power gating techniques with minimal impact of wake up time we can improve the sensor node lifetime significantly. This thesis shows that static power consumption can be reduced by anywhere from 20 % to 80 %, depending on the technique used. One can pair these techniques with energy harvesting systems that further maximize the mote oper-

ating lifetime. Even though these techniques are available in synchronous circuits, they usually come with power and delay penalties. Using power gating in the context of asynchronous systems translates in high power gains, with extremely little overhead and fast wake-up time.

In this thesis, we also present the analysis and design of a cryptographic system that is suitable for the WSN design space. Cryptography is a critical block within any WSN since encryption is arguably the first line of defense to protect the confidentiality of data over the wireless link. We implemented and benchmarked software, hardware, and hybrid AES implementations.

On one hand our cryptographic AES system can provide up to 30× performance over its software counterpart if a full hardware implementation is used. On the other hand, a complete hardware implementation is unattractive from the standpoint of lifetime due to increase in leakage current from the additional transistors.

If encryption throughput is not high-priority, a complete software AES implementation can offer a 3× increase in mote battery lifetime. By incorporating power gating techniques into our hybrid and full hardware designs, we can reduce the gap in battery lifetime to less than 66 %. If we compare the leakage of the required memory resources to the AES hardware implementation, the fully software implementation is only 10 % better than the full-hardware implementation. A hybrid implementation can offer 6× net performance improvement, and increases lifetime by 10 % over the full software counterpart.

Our measured and simulated results show that our microcontroller, ULSNAP, is a good fit for the WSN paradigm and snow how ULSNAP can help bring ubiquitous computing into our everyday life.

1.1 SNAP

Kelly, Ekanayake, and Manohar proposed a novel instruction set architecture (ISA) designed expressly for sensor networks called the SNAP ISA [30]. This ISA was originally to be implemented on a custom chip multi-processor. However, the event driven architecture of the SNAP ISA offered a unique opportunity for use in the WSN space. The SNAP ISA fits the WSN model by reducing the dynamic instruction count—oftentimes the dynamic instruction count for support code running on processors with more traditional ISAs processor is a two-fold the number of instructions that do useful application work [12]. To take advantage of this properly of the SNAP ISA, the designers developed the Sensor Network Asynchronous Processor (SNAP), which offered an attractive performance/energy tradeoff for the WSN space at the time it was fabricated.

Inspired by the high performance and low energy of SNAP, we present in this thesis the design of ULSNAP. The architectural of ULSNAP is a significant modification of the original SNAP design, which allows to run at a lower energy point with higher throughput. ULSNAP can deliver almost 100 MHz using less than 4.5 mW of power. The new functionality added to ULSNAP makes it an easy to use as a microcontroller for WSN nodes. Furthermore, the reduced power consumption in ULSNAP compared to its SNAP counterpart yields longer node lifetime while maintaining high throughput. The measured results of the original SNAP chip showed that SNAP can run at 4MHz and use 40 pJ on its most energy efficient configuration and run at a maximum throughput of 129MHz while using 250 pJ. In contrast, the ULSNAP implementation can run at 47MHz using only 29 pJ. While the SNAP chip, fabricated in 0.13 µm technology, proved itself a good candidate for WSN, the development of ULSNAP and the techniques presented in this

manuscript present a significant advancement in the capabilities provided by an asynchronous processor in a WSN node.

1.2 Contributions

The original contributions in this dissertation can be summarized as follows:

- Power gating in the context of asynchronous circuits. We present power gating techniques in the context of asynchronous circuits and we explain the minimum conditions and requirements to implement different power gating techniques in asynchronous circuits.
- Design, implementation, and measured results of ULSNAP. The circuits, architectures and methodologies presented in this manuscript push the limits of the state of the art by showing a microcontroller that is in the Pareto optimal front of the energy-delay curve of computation. Our *measured* results demonstrate an asynchronous high throughput low energy microcontroller that befit the WSN paradigm.
- Evaluation of the impact of using ULSNAP in Wireless Sensor Networks. We adapt a model for sensor network node lifetime to measure the impact of our proposed architectures and circuits on WSN node lifetime.
- Asynchronous Implementation of AES. Cryptography is a critical capability in the WSN space. In this thesis, we present the design and implementation of a fully asynchronous AES hardware block. Our circuit 8 pJ per bit while delivering 950Mbps. Furthermore, our implementation is the Pareto-optimal of the energy-throughput space compared to the best implementations in the literature.

- Analysis of Encryption in a WSN microcontroller. Encryption is arguably the first line of defense to protect data transmitted over a wireless link. In this thesis, we analyze the trade-offs of augmenting a microcontroller with hardware, software, and hybrid configurations of an AES encryption engine, as well as the implementation details of an AES encryption engine on a WSN microcontroller.

CHAPTER 2

BACKGROUND

Sometimes you are in sync with the times,
sometimes you are in advance,
sometimes you are late

Bernardo Bertolucci

We can categorize digital circuits as being synchronous or self-timed (aka asynchronous). Synchronous circuits rely on a global signal called a *clock* to implement sequencing and determine when data can be sampled. Self-timed circuits use handshaking protocols to provide synchronization between processes and uses various methods to determine the validity of data. Although many self-timed circuits families exist, in this thesis we use mostly Quasi Delay-Insensitive (QDI) circuits introduced by Martin [39]. A survey of other self-timed methodologies and circuit families can be found in [17].

2.1 Quasi Delay-Insensitive (QDI) Circuits

Most of the circuits described in this thesis were built using QDI circuits derived using Martin synthesis [39]. QDI circuits are a subset of asynchronous circuits since they operate without a global clock.

While synthesizing QDI circuits, the design specification is first expressed in the Communicating Hardware Processes (CHP) language. The synthesis procedure decomposes a CHP program into many fine-grained hardware processes operating

in parallel. These processes synchronize by communicating tokens over delay-insensitive channels that are implemented using a delay-insensitive protocol.

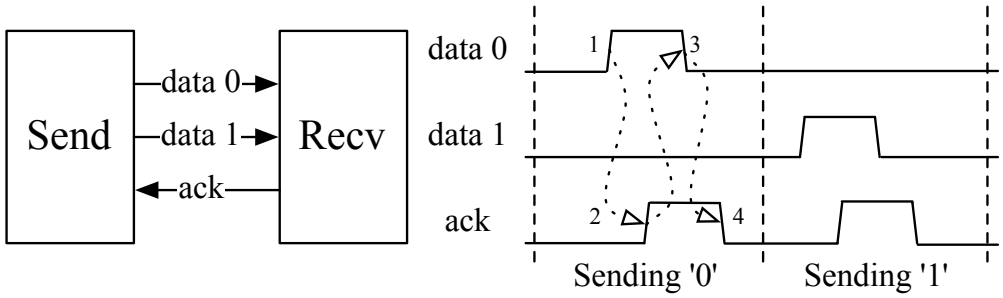


Figure 2.1: Four-Phase Handshake

A single-bit channel communication is shown in Fig. 2.1. Here, the processes synchronize using a delay-insensitive protocol we refer to as the dual-rail four-phase handshake protocol [68]. To transmit data, the sending process asserts either the true (data1) or false (data0) line, which the receiving process acknowledges, after which the channel resets (3,4).

2.2 Synthesis of QDI circuits

Martin describes a synthesis method for QDI circuits [39]. Fig. 2.2 shows the design flow for circuits created with this method. The overall design specification of a circuit is first expressed by CHP This first CHP program is often referred as *Sequential CHP*. A short summary of CHP can be found in Appendix A. After a series of CHP to CHP transformations, the CHP is broken down into a set of simple processes running in parallel. This set of CHP programs is often referred to as *Decomposed CHP*. The transformation of a sequential specification to the final concurrent system is done using semantics preserving transformations. Therefore, if we know that the original sequential specification is correct the final concurrent

implementation is correct as well [40]. The main program transformation methodologies include *Process Data Decomposition* [39], *Projection* [37], and *Pipelined mutual exclusion* (PME) [36].

Once decomposed, a CHP undergoes a series of syntactic translations that replace all channel communication actions with a handshake protocol. At the same time, all variables assignments are replaced by boolean-valued expressions in a process known as *Handshaking Expansion*. The resulting program is a subset of CHP using only boolean-valued expressions. This sub-language is often referred to as handshaking expansion (HSE) as well. The HSE is then transformed into a *Production Rule Set* (PRS) by following a *Production Rule Synthesis* that consists on three main steps: state assignment, guard strengthening, and symmetrization.

A production rule (PR) takes the form: $G \mapsto S$, where G is a boolean expression called the guard, and S is the boolean assignment. Typically, a PR corresponds to a pullup or pulldown switching network, depending on whether the boolean S was true or false, respectively. For example, program 2.1 represents the production rule for the pullup network of node c.

Program 2.1 A sample production rule representing pullup network of node c

$$a \wedge b \;\rightarrow\; c\uparrow$$

A production rule can be transformed into a digital netlist by making sure that all rules in the PRS are CMOS-implementable. For the PRS to be CMOS-implementable, all variables used in the pulldown must not be inverted and all variables in the pullup must be inverted. The process of converting a PRS into one that is directly implementable in CMOS is called *bubble reshuffling*.

As an illustrative example we discuss how to express a pipeline stage imple-

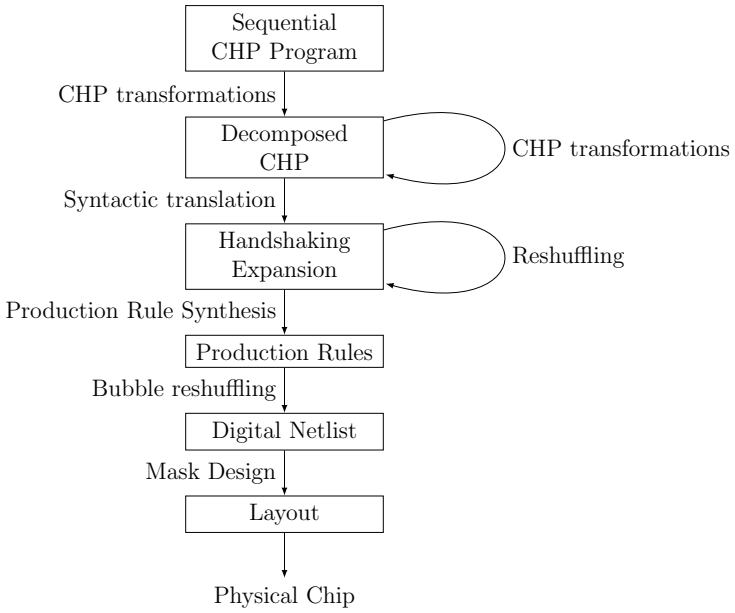


Figure 2.2: Synthesis of Asynchronous Circuits

Program 2.2 Buffer performing computation $g(f(\cdot))$

$$S \equiv *[\ IN?x;\ \ OUT!(g(f(x)))\ ]$$

menting a complex function $(g \cdot f)(x)$. This function is described by the CHP program 2.2.

The stage, S, accepts a data token, x, on an input channel, IN. The output of the pipeline stage on the output channel OUT is a data token representing the computed value of the function $g(f(x))$.

We can first break the complex $g(f(x))$ function into two functions as shown in program 2.3.

Program 2.3 Buffer performing computation $g(f(\cdot))$

$$S \equiv *[\ IN?x;\ \ y := f(x);\ \ z := g(y);\ \ OUT!(z)\ \ ]$$

Furthermore, we can remove the unnecessary sequencing on program 2.3 by projecting on variable x as shown on program 2.4

Program 2.4 Projecting on variable x,z

$$S \equiv \ *[IN?x;\ (\ Y!f(x)\ \|\ Y?y\ );\ z := g(y);\ OUT!z\ ]$$
$$S \equiv \ *[IN?x;\ Y!f(x)] \ \|\ *[Y?y;\ Out!g(y)]$$

The resulting process is shown in program 2.5:

Program 2.5 Decomposed buffer

$$S \equiv \ *[IN?x;\ Y!f(x)] \ \|\ *[Y?y;\ Out!g(y)]$$

In this case, the decomposition was straightforward. In the case of a complex circuit such as an encryption unit, the decomposition will go through several iterations, each producing a valid implementation of the original CHP implementation.

2.3 Data Encoding

QDI circuits rely on the ability of two connected processes to detect when there is valid data being transmitted. This detection operation must be possible no matter what the delay between the two processes is. This operation of transmitting channels over delay-insensitive channels is achieved using delay insensitive encodings [68]. The 1ofN encoding achieves this by encoding bits information with N rails [20]. To communicate data between our processes, we typically use $1of2$ or $1of4$ codes for data and 1ofN codes for control.

A $1of2$ code transmits a single bit of information with two wires, the *true* wire and the *false* wire. For a $1of2$ channel X, we name these wires $X.t$ and $X.f$ respectively. We use a handshake protocol as shown in Fig. 2.1 to transmit data over these delay insensitive channel. When the condition $X.t \vee X.f$ holds, we say that channel X is *valid*. When the condition $\neg X.t \wedge \neg X.f$ holds, we say that the

channel is neutral. The state when $X.f \wedge X.t$ holds is invalid and should never be reached. The $1of2$ encoding is also known as dual-rail.

In the 2-bit case, we can use a $1of4$ encoding [68]. A $1of4$ channel uses 4 wires to transmit data. Each rail represents one value. Therefore, 4-wires encode 2 bits worth of data. Similar to the $1of2$ channel, the state where $Y.d[0] \vee ... \vee Y.d[3]$ holds means data is valid and ready to be processed. The condition where $\neg Y.d[0] \wedge ... \wedge \neg Y.d[3]$ holds means a $1of4$ channel is neutral. The main advantage of using a $1of4$ encoding is that only 1 rails toggles to transmit 2 bits of data, hence being more energy efficient than a bundle of two $1of2$ channels. In other words, even though the number of wires in a $1of4$ channel is the same as in a $2x1of2$ bundle, only half as many of the switch during a transmission.

When we want to transmit M bits, we can bundle multiple bits into a $Mx1of2$ channel or a $\frac{M}{2}x1of4$ bundle. The advantage of using this bundle is that a single acknowledge signal e is shared among the multi-bit data slice. Similar to the 2-bit channel a $\frac{M}{2}x1of4$ bundle is preferred since only half as many wires switch during an data transmission compared to its $Mx1of2$ counterpart.

2.4 Properties of QDI circuits

The data-driven nature of QDI designs allows a circuit to idle without switching activity when there is no work to be done. Another advantage of QDI circuits is the capability for correct operation in the presence of continuous and dynamic changes in delays [42].

This style of pipelining is entirely data-driven. Without an input token on the

IN channel, no computation takes place and the circuit is idle, consuming only leakage power. Composing pipelines out of stages similar to S is as simple as connecting stages together at channel boundaries, e.g. OUT of one stage to IN of the next stage, and so on. Synchronization and flow control is handled locally on a channel-by-channel basis.

This local synchronization behavior also enables average-case system performance. The timing of QDI circuits is exemplified in Fig. 2.3, in contrast to synchronous systems, which must define their clock period by the slowest pipeline stage, the performance of an asynchronous system is set by the critical path of *active* pipeline stages or functional units.

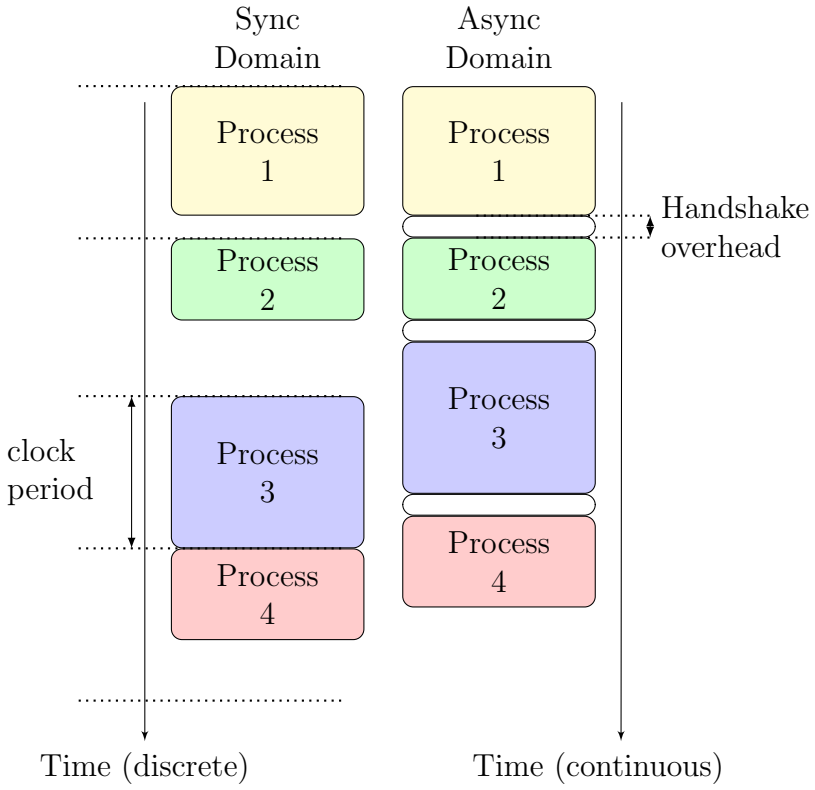


Figure 2.3: Synchronous and Asynchronous Time Domains

The average performance of an asynchronous circuit is thus governed by the most often exercised execution paths. This property allowed us to optimize rarely

used functional units for energy efficiency instead of trying to meet aggressive performance targets to meet our timing targets. While synchronous designers can implement complex functional units as multi-cycle units, they must account for the resultant synchronization and control overheads. In an asynchronous pipeline all synchronization is handled by the local handshakes, as described earlier—there is no additional overhead aside from the momentary reduction in performance when a slow functional unit is exercised.

In addition to being naturally data-driven, QDI circuits operate correctly in the presence of arbitrary wire or gate delays or other continuous and dynamic changes on delays. Sources of such local delay variations may include temperature, supply voltage fluctuations, process variations, noise. This robustness to delay translates to robustness to variations in the fabrication process, operating voltage and temperature.

QDI circuits are able to function at different voltages, and can even function with dynamic fluctuations on the supply voltage. This ability to function at multiple voltage levels means that a single circuit can be used at high voltage for high performance and at a low voltage for energy efficiency and low static power.

CHAPTER 3

PERFORMANCE OF SENSOR NETWORK MICROCONTROLLERS

If you want to find the secrets of the universe, think in terms of energy, frequency and vibration

Nikola Tesla

3.1 Energy Performance Pareto Optimal Set

As of this writing, the most important figures of merit of a microcontrollers are: *energy* [J] and *throughput* [Hz]. The throughput measures the number of tasks a computer can perform within a period of time and it is usually measured by the cycle time [Hz], instructions per second [MIPS], or tasks per second. Energy refers to the amount of electrical energy required by the system to complete a task. The energy of the system is given by Eq. 3.1, where P is the power in Watts and E is the energy in Joules.

$$E = \int_0^t P \, dt \tag{3.1}$$

If we assume a relatively constant power draw, then we can compute the transferred energy by using Eq. 3.2. On one hand, reducing the time to complete a task reduces the energy of the system. On the other hand, the most common practices to reduce the cycle time are: increasing the system voltage, which has a quadratic relationship to the energy, and reducing the effective resistance of each circuit, which is inversely proportional to the energy transferred to the system.

$$E = VI \cdot t = \frac{V^2}{R} \cdot t \tag{3.2}$$

These design constraints oftentimes leads to designers aggressively reducing energy during active computation at the cost of performance [65]. It is very difficult to achieve energy reduction without having a substantial cost cost in performance. Our goal is to achieve high performance and low energy, since the even increasing application requirements and high cost of communication requires an increase on the computation performed locally.

Instead of evaluating a global function that encompass both objectives, we use *Pareto* optimality, which describes a set of efficient solutions with multiple (oftentimes conflicting) objective functions. In the Pareto optimal set of design configurations, each member has the non-dominated property: for each set member, there are no members of the universe set of configurations which outperform it. While Pareto optimality has been used for a long time in economics [15], computer science and other engineering fields have also benefited from it.

Fig. 3.1 shows an example of the energy-performance distribution of a set of elements. The red line represents the Pareto optimal set of all the elements in the space. Each of the elements on the Pareto optimal set is better in throughput, lower in energy or both than other elements in the set. The Pareto front is also known as the Pareto frontier, the Pareto optimal, and the Pareto dominant set.

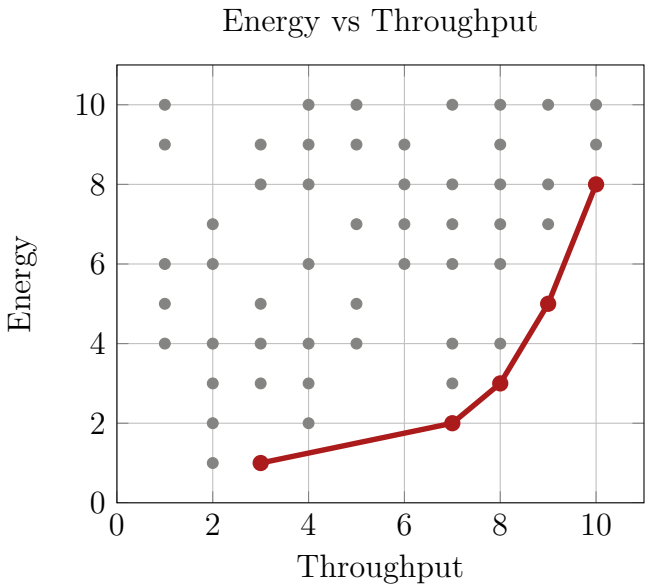


Figure 3.1: timal front of the energy vs. throughput space. The red line represents the pareto optimal set of the element space

3.2 Mote Lifetime Model

We adapted an analytical model that captures how the architecture of a generic mote system affects battery lifetime from the work of Jung *et al.* [27]. The authors of the original model propose a semi-Markov chain formulation of the power state transitions. In Sec. 6.4, we augment this model to analyze the impact of multiple architectures of a cryptographic system on the battery lifetime. Our adapted model has the following properties:

- Ergodicity: The mean value of all quantities is known by observation of a large enough sample.
- Event arrivals follow a Poisson distribution.
- Sensing, processing and transmission times are independent and identically distributed with the same arbitrary distribution.

Fig. 3.2 shows our semi-Markov chain model for a generic cryptographic WSN mote's power states: Sensing (S_1), Processing (S_2) and Transmit or TX (S_3).

In our model, we assume the processing and communication steps finish execution as fast as possible, i.e. they never transition to a low-energy mode when data is available to compute. α represents the probability that we will transmit data after processing. Fig. 3.3 shows a power trace for a sample WSN execution. X, Y, and Z represent the average *sensing time*, the average *processing time*, and the average *communication time* respectively, and the inter-arrival time is $1/\lambda$.

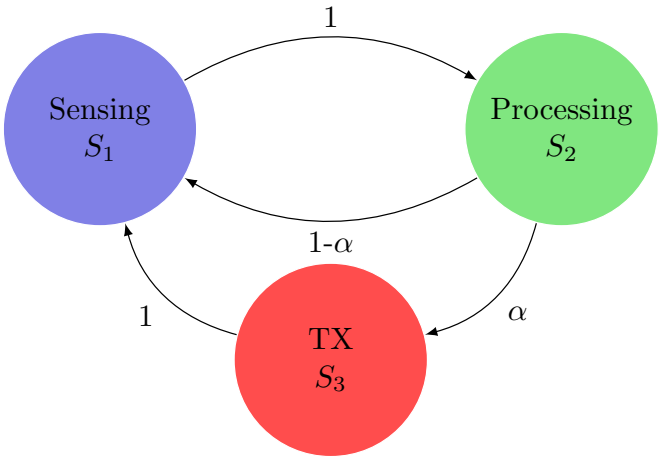


Figure 3.2: Semi-Markov Chain for cryptographic WSN Mote

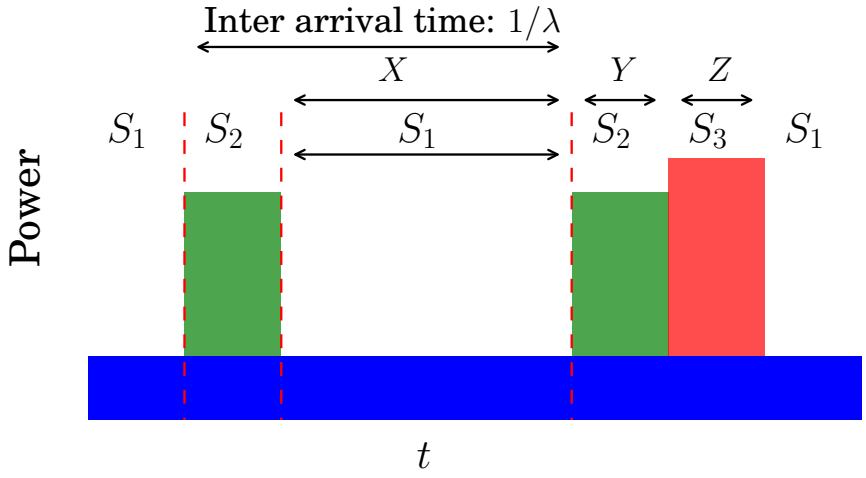


Figure 3.3: Mote Power Profile

The total energy spent across all states is defined in Eq. (3.3). Note that E_{total} cannot exceed the energy provided by the mote battery source. Each E_i represents the energy consumption of a particular state, where p_i is the steady state probability of being in state S_i, t_i is the total time spent in S_i, and P_i is the power consumed by S_i.

$$E_{\text{total}} \geq E_1 + E_2 + E_3 \tag{3.3}$$

$$E_i = t_i P_i \tag{3.4}$$

Given a long enough time period, T, the total time spent at state S_i can be approximated as $\lim_{t\to\infty} t_i = p_i t$. Therefore, $E_i = p_i t P_i$. Let π_j be the *stationary probability* of the Markov chain, which is the frequency of visiting each state over an infinite execution. p_{ij} is the probability of a transition from S_i to S_j. π_j can be interpreted as the proportion of transitions into state j.

$$\pi_j = \sum_i \pi_i p_{ij}, \ \sum_j \pi_j = 1 \tag{3.5}$$

The probabilities p_{ij} can be obtained from our model in Fig. 3.2, and the equations can be written in matrix form, where row indices represent source states and column indices represent end states:

$$\begin{bmatrix} \pi_1 & \pi_2 & \pi_3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1-\alpha & 0 & \alpha \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \pi_1 & \pi_2 & \pi_3 \end{bmatrix} \tag{3.6}$$

We can then solve the resultant system of equations represented in Eq. (3.6) and the latter half of Eq. (3.5) to obtain the following:

$$\pi_1 = \pi_2 = (2+\alpha)^{-1} \qquad \pi_3 = \alpha(2+\alpha)^{-1} \tag{3.7}$$

The Markov chain model allows us to express p_i as follows, where μ_i is the mean time spent in S_i before making a transition—X, Y, and Z, respectively:

$$p_i = \frac{\pi_i \mu_i}{\sum_j \pi_j \mu_j} \tag{3.8}$$

Equations (3.3), (3.4), and (3.8) allow us to solve for t_{life} as a function the average time spent in each state and the probability of state transition from Processing to Transmit, α.

$$E_{\text{total}} \geq t \frac{\bar{\mu_1} P_1 + \bar{\mu_2} P_2 + \alpha \bar{\mu_3} P_3}{\bar{\mu_1} + \bar{\mu_2} + \alpha \bar{\mu_3}} \tag{3.9}$$

$$t_{\text{life}} \leq \frac{E_{\text{total}}(X + Y + \alpha Z)}{\bar{\mu_1} P_1 + \bar{\mu_2} P_2 + \alpha \bar{\mu_3} P_3} \tag{3.10}$$

The state S_1 corresponds to the idle state where the activity factor is quite low in comparison to that of states S_2 and S_3. In state S_1, a QDI microcontroller is effectively clock-gated due to its data-driven nature. As such, the static power consumption of is a reasonable proxy for the power consumption in S_1. Assuming that $X \gg Y, Z$, we approximate the average sensing time X as the inter-arrival time $1/\lambda$. We can rewrite Eq. (3.10) as:

$$t_{\text{life}}(\lambda) \leq \frac{E_{\text{total}}(1 + \lambda(Y + \alpha(\bar{\mu_2}))}{P_1 + \lambda(\bar{\mu_2} P_2 + \alpha(\bar{\mu_3} P_T))} \tag{3.11}$$

CHAPTER 4

ASYNCHRONOUS POWER GATING

Beware of little expenses. A small leak will sink a great ship

Benjamin Franklin

In this Chapter, we present a summary of power gating techniques in the context of asynchronous circuits. Furthermore, we explain the minimum conditions and requirements to implement power gating in asynchronous circuits and present detailed implementations of different power gating schemes.

Power gating is perhaps the single most important tool circuit designers have to combat leakage. These techniques increase the effective resistance of leakage paths by adding sleep transistors between logic transistor stacks and power supply rails. Power gating also enjoys many of the benefits obtained from transistor stacking. Oftentimes, these power gating or sleep transistors are shared among multiple logic stacks to reduce the number of leakage paths as well as area overheads. Sharing the transistors effectively creates two new power nets: Gated-Vdd (*gvddv*) and Gated-Ground (*gvssv*), which replace V_{DD} and GND for power-gated logic stacks. *gvddv* is connected to V_{DD} using a head sleep transistor and *gvssv* is connected to GND using a foot sleep transistor.

4.1 Pseudo-Static Logic Overview

The production rules for an operator with a pullup network expression *pun*, pull-down network expression *pdn*, and output node z are shown in Program 4.1:

Program 4.1 Pullup and Pulldown network of a CMOS operator

$pun \rightarrow z\uparrow$
$pdn \rightarrow z\downarrow$

Such an operator is non-interfering and *combinational* if $pun \equiv \neg pdn$. The weaker constraint of $pun|\ pdn \equiv \mathbf{true}$, denotes a non-interfering, *dynamic* operator. Adding a staticizer to the output node, z, of a dynamic operator ensures the output is always driven. Such an operator is known as a *pseudo-static* gate. An asynchronous circuit is comprised of a mix of combinational and these pseudo-static gates and operators.

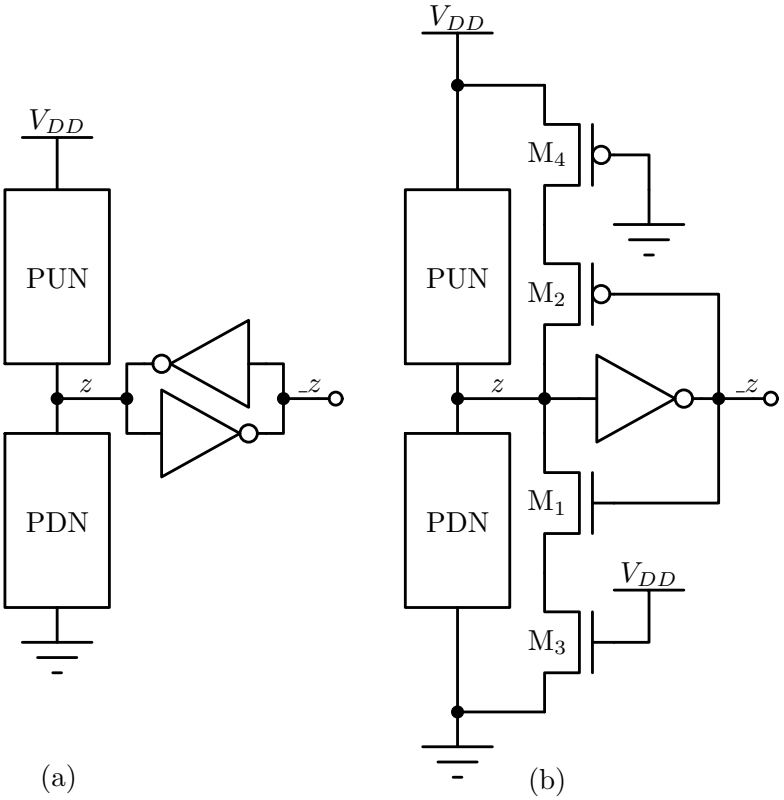


Figure 4.1: (a) Pseudo-Static CMOS Gate, (b) Weak Feedback Inverter

An implementation of a generic pseudo-static operator is shown in Fig. 4.1a. The statizicer consists of two cross-coupled inverters attached to node z. Note

that there is always opposition to any change in z due to the feedback inverter. To ensure correct operation, the transistors of the feedback inverter must be sized to be weaker than the logic stacks of the operator. Furthermore, the feedback transistors add parasitic capacitance to the output node. To mitigate this effect, each feedback transistor is split in two, as shown in Fig. 4.1b. The feedback stack now consists of a minimum sized transistor closer to the output, $M_1(M_2)$, and a long transistor closer to the power rails, $M_3(M_4)$. In order to reduce the load on node $_z$, the gates of the long transistors, $M_3(M_4)$, are usually connected to $V_{DD}(GND)$ or to $Reset(\_Reset)$.

4.2 Non-State Preserving Power Gating

Non-state preserving techniques destroy the state by allowing internal nodes to uniformly drift towards one of the power rails. The general class of power-gating techniques has various implementation methodologies that include *Cut-Off* (CO), *Multi-Threshold* (MTCMOS) [21], *Boosted-Gate* (BGMOS), and *Super Cut-Off* (SCCMOS).

The primary disadvantage of these techniques is that the state of internal nodes is lost. Fig. 4.2 shows, the implementation of the Cut-Off power gating technique using a foot sleep transistor inputs to the first stage while idle are logic, and the output.

Any of the previously discussed non-state preserving techniques can be applied to pseudo-static logic. However, waking up a circuit without resetting all its pseudo-static elements into known, safe states could result in incorrect circuit behavior, or even the potential for stable short-circuits between power rails.

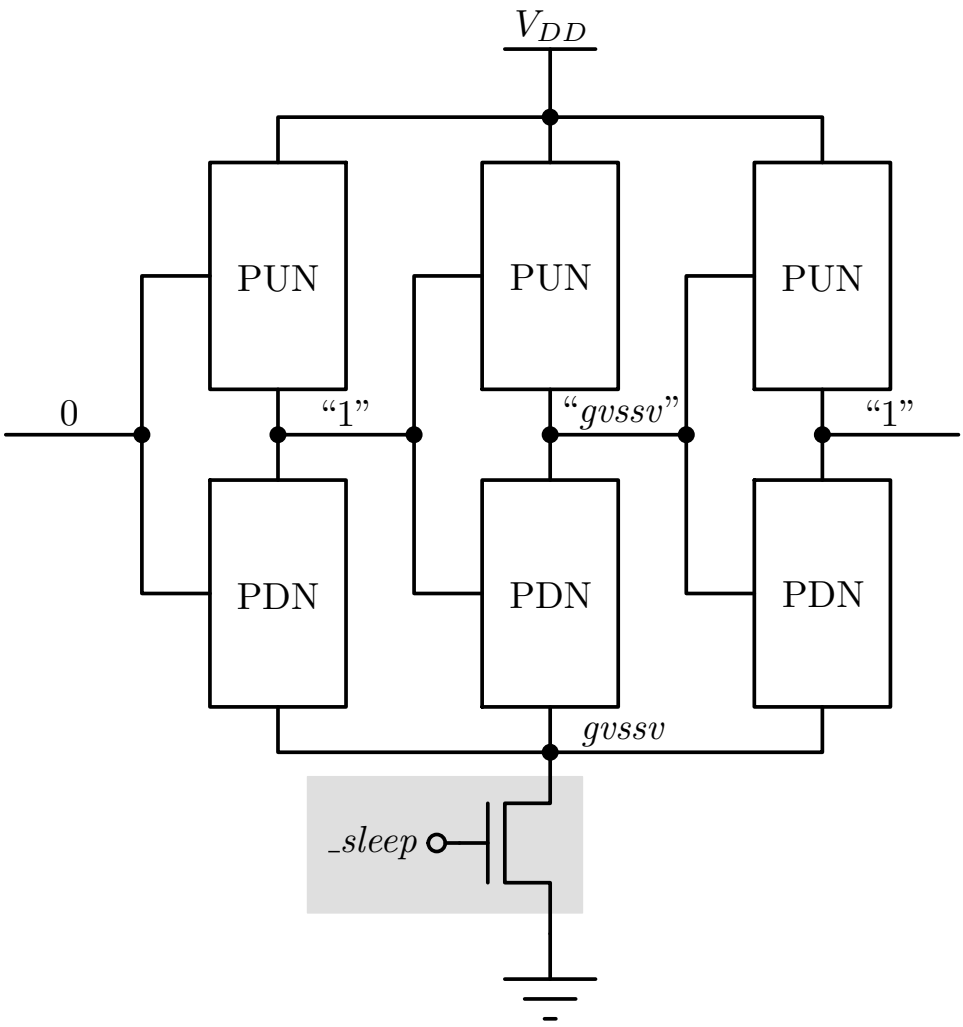


Figure 4.2: Cut-Off (CO) power gating using a foot sleep transistor, which is shared by several logic blocks. The output nodes drift to *gvssv*, which itself drifts towards V_{DD}.

This problem is not unique to power gating—in fact, it is a concern during the initial power up of asynchronous circuits, which use pseudo-static gates. Fortunately, the addition of reset transistors to initialize the appropriate circuit nodes is a viable solution. In the case of power up, the signals which drive the gates of these reset transistors are generated off-chip. However, initial power up is a global event. As the off-chip environment is unaware of the entire internal state of the chip, generating reset signals for each individual power gated circuit off-chip would prove to be practically impossible, even just considering package pins as a

limitation.

To ensure correctness and safe operation, each power gated circuit requires its own self reset circuitry. In our asynchronous design methodology, we use transistors both in series and in parallel with pullup and pulldown stacks. To control the parallel and series reset transistors, we use *pReset* and *sReset* signals and their complements, respectively. While the order and delay between asserting *pReset* and *sReset* is flexible, *pReset* must be deasserted before *sReset* to prevent any short circuits between power rails. A typical reset sequence is as follows:

1. Assert *pReset*, *sReset*, and their complements and hold them until all the circuit output nodes have been charged to their appropriate safe states.
2. Deassert *pReset* and its complement.
3. Deassert *sReset* and its complement.

Note that in order for the self reset circuit to be QDI, it would have to instrument every output node in order to determine whether or not it has reached the appropriate safe state during step 1 above. This endeavor quickly becomes very costly in transistor count, area, complexity, and power. A similar argument applies for determining the appropriate delay between steps 2 and 3 above. As such, the self reset circuit we propose is not QDI, but instead relies on the timing assumption that a delay line, tailored to the circuit being reset, is sufficient to guarantee safe reset of all internal circuit nodes. Again, a similar argument involving a delay line between steps 2 and 3 applies.

Upon deasserting the *sleep* signal, i.e. waking up the circuit, the self reset circuitry will assert *sReset* and *pReset* in that order, then deassert them in reverse

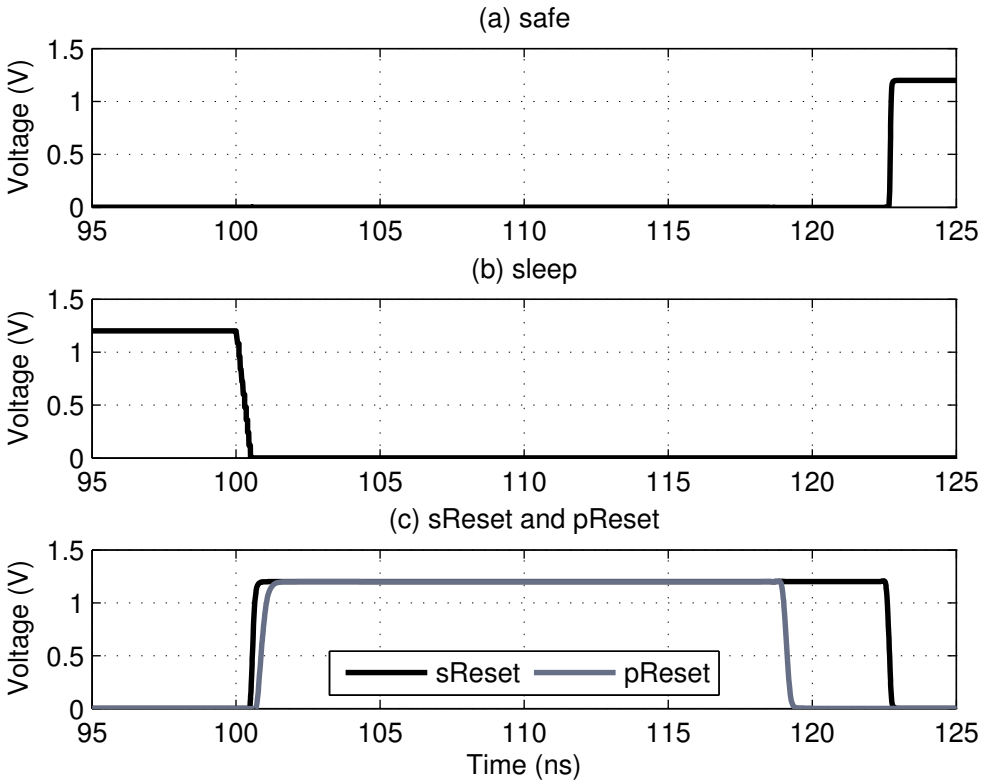


Figure 4.3: Self reset circuit behavior immediately after *sleep* goes low.

order as seen in Fig. 4.3. The timings between these transitions are controlled by delay lines. Note that *pReset* should be held long enough to account for the charge/discharge latency of the local supply rails—i.e. *gvssv*—and the worst case reset latency. Depending on process variations, it may be desirable to further increase the hold time of *pReset*. In fact, it is advisable to layout the delay line as close to the logic as possible in order to replicate localized systematic process variations. Once the self reset sequence is complete, a *safe* signal is raised, as seen in Fig. 4.3a.

From the time the circuit has been power gated until the circuit completes its internal self reset, the outputs of the gated circuit are undefined. If the rest of the pipeline is operating, these undefined outputs should not corrupt the rest of

the system, particularly pipeline stages which have been fully woken up. This impacts both the pipeline stage inputs—through acknowledge signals—and outputs—through data signals. *Isolation circuits* are introduced to make sure that all output signals from the power gated block remain in a well-defined state. Adding isolation circuits to the input of a stage prevents signals from interfering with the self reset of a stage, and isolation circuits on the output prevent any glitches from propagating to other pipeline stages during the self reset stage.

4.3 State Preserving Power Gating

State preserving power gating techniques reduce leakage while retaining state. The tradeoff between these techniques and non-state preserving techniques is that they are not as effective as at reducing leakage currents.

The two main state preserving power gating techniques are *Variable Threshold* (VTCMOS) and *Zig-Zag Cut-Off* (ZZCO) [21] VTCMOS has the disadvantage of requiring a bias voltage generator, as well as the use of triple well process.

Our state preserving power-gating scheme is based on the Zig-Zag Cut Off (ZZCO) power gating, it offers a good tradeoff between power savings and performance degradation for this class of power gating. As in non-state preserving techniques, ZZCO introduces two power nets: Gated-Vdd (gvddv) and Gated Ground (gvssv). Rather than gating every logic state in the same fasion, the selection of the head or foot transistor is governed by the desired logic level of the output node. As shown in Fig 4.4, *gvddv* i and *GND* are used as power rails for logic blocks with a logic 0 output when idle and V_{DD} and *gvssv* for blocks with with logic 1 output when idle.

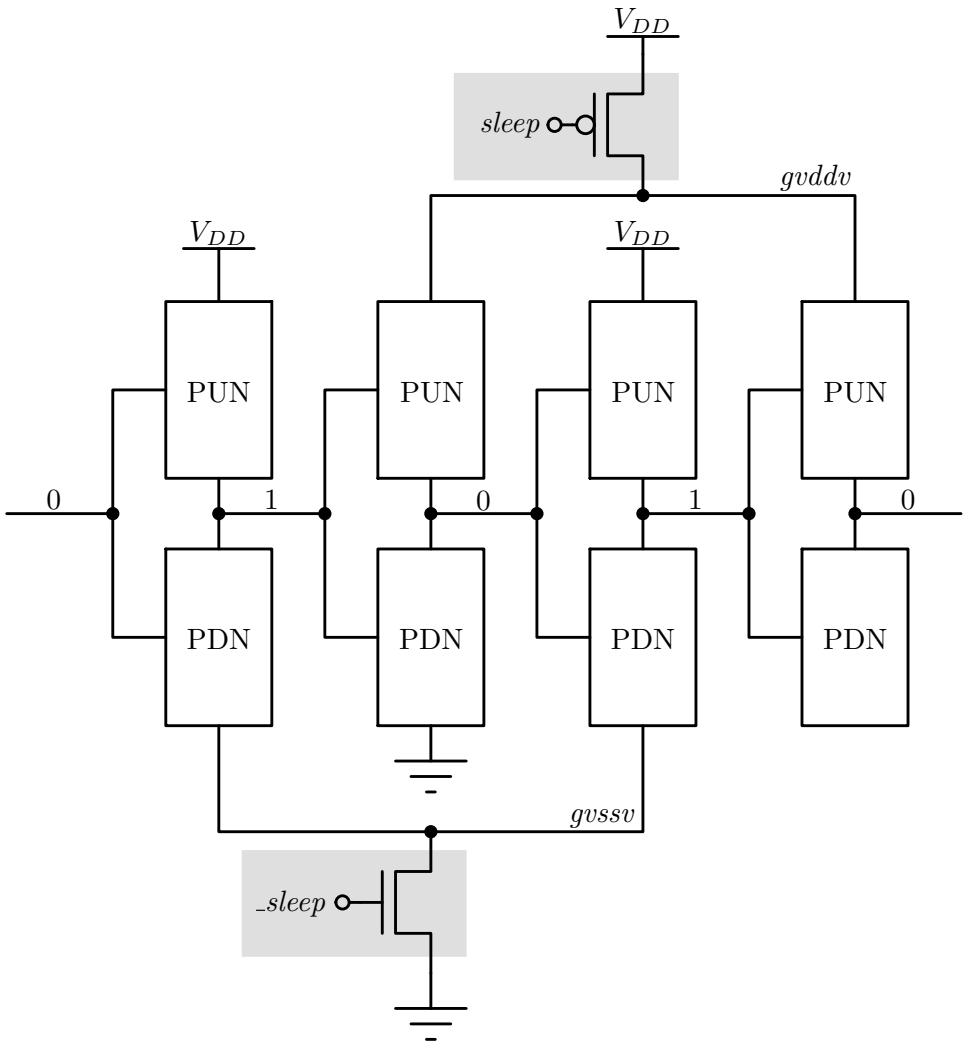


Figure 4.4: Zig-Zag Cut-Off (ZZCO) using a pair of sleep transistors, which are shared between several logic blocks. The configuration of sleep transistors restores the otuptu nodes to the appropriate idle state values.

For asynchronous circuits, in idle mode, we know there are no inputs and that all logic blocks have finished computation. Therefore, each individual logic block is waiting for data. By analyzing the handshaking expansions of each process, we can ascertain the value of most signals in the idle state. One exception involves the case of two-phase handshakes where the number of handshakes is not guaranteed to be even. Nevertheless, for most cases, we can use Zig-Zag power gating by connecting all the logic blocks whose output is logic 1 to *gvssv* and all the nodes

whose output is logic 0 to *gvddv*.

In order to efficiently power gate pseudo-static operators, we gate the forward inverter of the staticizer in addition to the logic stacks depending on the idle state output of the logic. Essentially, pseudo-static Zig-Zag Cut-Off (ZZCO) power gating adds sleep transistors to the logic stack and the feedback transistors of pseudo-static operator shown in Fig. 4.1b.

We can reduce the leakage through the feedback inverter by connecting the gates of M_3 and M_4 to *gvddv* and *gvssv*, as shown in Fig. 4.5a. Alternatively, their gates could be connected to the sleep signal directly, as in in Fig. 4.5b, but the area penalty would be high because the sleep signal would need to be routed individual staticizers, as opposed to just the shared sleep transistors. We refer to the technique of driving the gates of M_3 and M_4 with *gvddv* and *gvssv* as Zig-Zag Cut Off with Weakened Staticizers (ZZCO-WS).

Note that the only difference between ZZCO and ZZCO-WS is between which signals drive the gates of M_3 and M_4. Thus, the area overhead for implementation of ZZCO-WS versus ZZCO is negligible, as all the supply nets—i.e. *gvssv*, *gvddv*, *GND*, and V_{DD}—are locally accessible to each layout cell.

We chose Cut-Off (CO) and Zig-Zag Cut-Off (ZZCO) as our non-state holding and state holding power gating techniques, respectively, as neither requires bias voltages or multiple-well capabilities. The complexity and trade-offs of bias voltage generation made it unattractive to implement. For example, even though SCCMOS offers better leakage reduction versus CO, the current draw of the bias generation circuits make SCCMOS viable for only large circuits. In our 90nm technology, a switched capacitor bias generator, based on the baseline generator from [66],

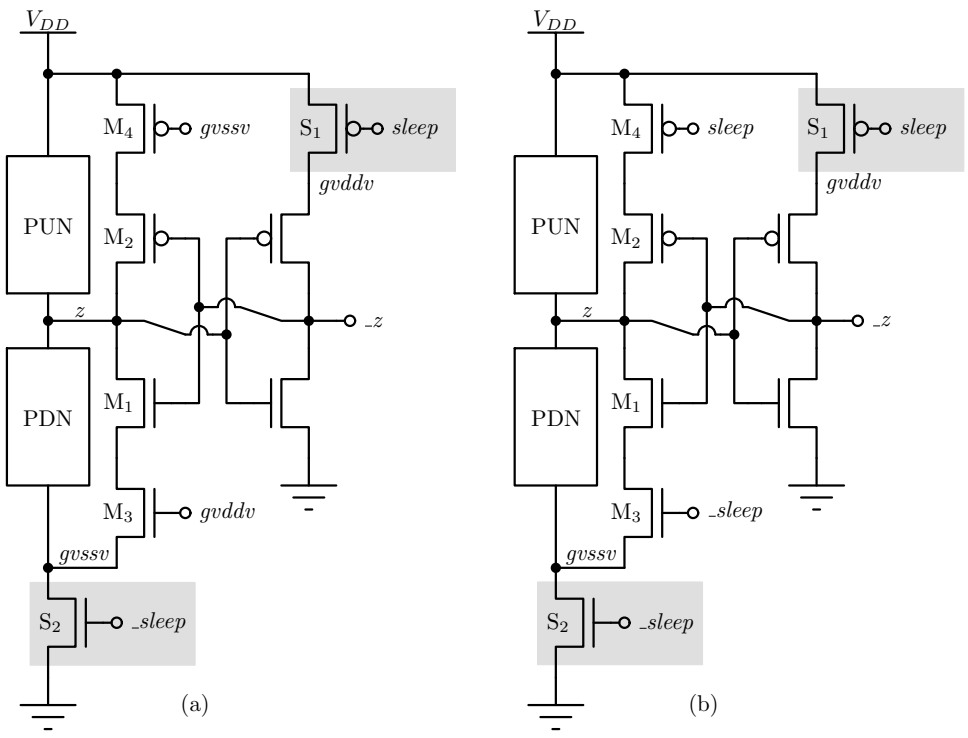
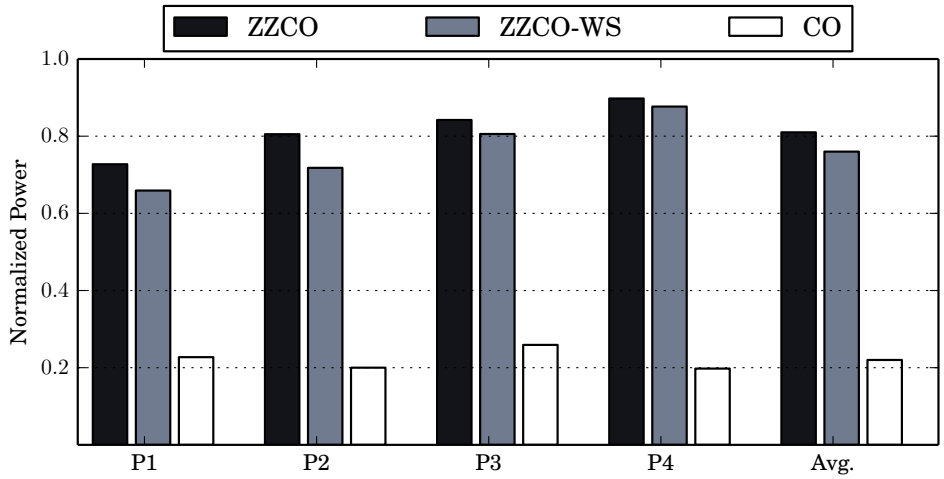


Figure 4.5: Zig-Zag Power Gating with Weakened Staticizers (ZZCO-WS) using (a) Virtual Power Rails or (b) Sleep Signals

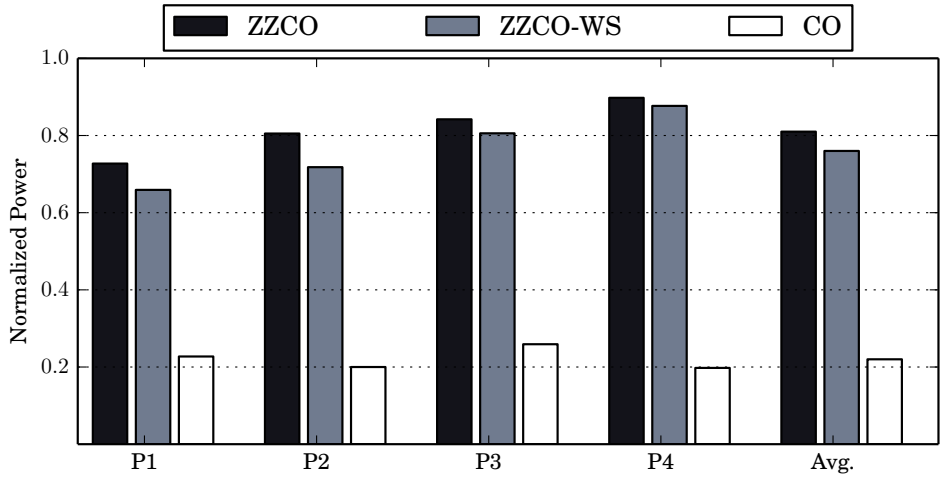
consumes an average of 116 µW. As such, power gating schemes which require on-chip bias generation with conventional circuits are inappropriate for any ultra-low power applications with static power in the sub-microwatt regime.

As seen in Fig. 4.6, ZZCO reduces leakage power by an average of 20%. If we weaken the staticizers (ZZCO-WS) during idle time as discussed in section 4.3, we save an additional 5%. However, the maximum savings in power come from using CO power gating, as it offers a 82% reduction in leakage power on average. The power reductions from ZZCO and ZZCO-WS are similar in both 65nm and 90nm technologies; however, CO power gating saves an additional 8% of static power in 65nm versus 90nm.

As for performance, Fig. 4.7 shows the degradation for different pipelines.



(a) 90 nm 25 °C

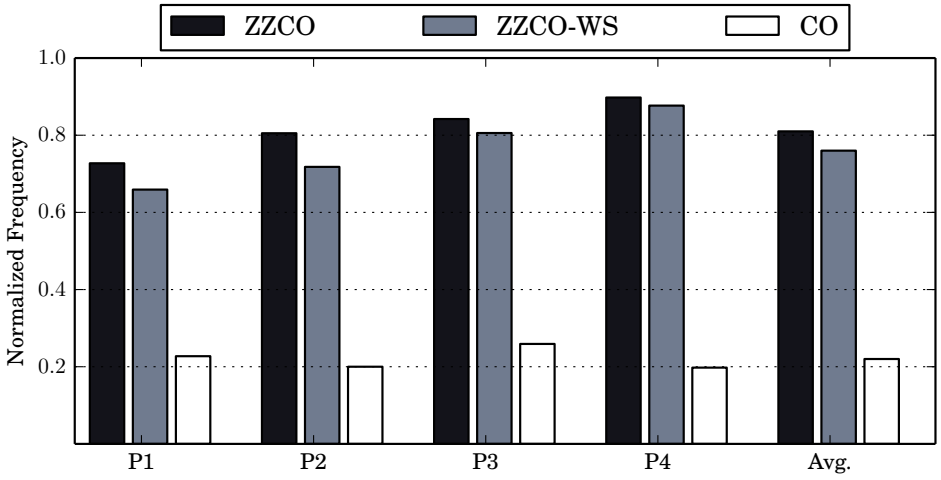


(b) 65 nm, 25 °C

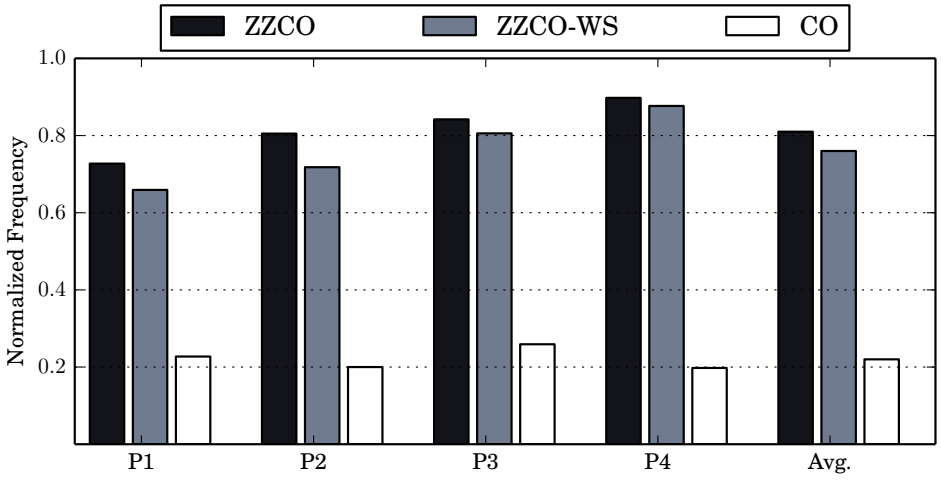
Figure 4.6: Static power consumption of 4 pipelines. Each pipelined is power gated in isolation, and results are normalized to a baseline implementation with no power gating.

ZZCO has the most pronounced effect on average operating frequency with a 29% degradation in 90nm and a 28% degradation in 65nm. ZZCO-WS is slightly better with degradation of 24% and 21% in 90nm and 65nm, respectively, and CO has the least impact of the three schemes, averaging a 23% degradation in 90nm and a 20% degradation in 65nm. Using *gvssv* and *gvddv* to drive the gates of the series transistors instead of *GND* and V_{DD} weakens the feedback stack, reducing

leakage as well as the opposition to changing the output node z, which origin of the performance improvements.



(a) 90nm, 25 °C



(b) 65nm, 25 °C

Figure 4.7: Average operating frequency of 4 different pipelines. Each pipelined is power gated in isolation, and results are normalized to a baseline implementation with no power gating.

CHAPTER 5

ULSNAP: ULTRA-LOW ENERGY EVENT DRIVEN MICROCONTROLLER FOR SENSOR NETWORK NODES

It is not what happens, but how you react to it that matters

Epictetus

WSNs are comprised of many small, low cost nodes or "motes" that gather, process, and propagate data about their surrounding environment. Typical motes are comprised of environmental actuators, sensors, a microcontroller, a radio or other communication interface and an energy supply such as a coin-cell or thin-film batteries. The on-board microprocessor handles local data processing and control tasks. Many motes also have the capability of entering a low-energy sleep state and are oftentimes paired with an energy harvesting system in order to maximize mote operating lifetime.

Mote deployment lifetimes can exceed several months, making battery life a crucial metric in this design space. Fortunately, most sensor network applications are bursty, e.g. only engaging in active execution when sensor data is available and then returning to a quiescent state. The idle or "sleep" state is oftentimes significantly longer than the execution period, so minimizing power during this idle phase is of paramount importance. On the other hand, increasing application complexity requires greater computational power, forcing more aggressive peak performance targets for sensor nodes. The high cost of wireless communication also contributes to increased demand for performance—computing results locally at a sensor mote is often a better system-level trade-off than wirelessly transmitting raw data [1]. Hence, a mote equipped with a powerful processor can increase the

mote lifetime and improve the performance per task.

In order to achieve these goals and fit the bursty computation paradigm, we implemented the Ultra-low Power Sensor Network Asynchronous Processor (ULSNAP). ULSNAP was optimized to be event-driven at both the architectural and circuit levels. We make use of the QDI family of self-timed, i.e. asynchronous, circuits described in Chapter 2.1, which are particularly well-suited for event-based computation as they follow a data-driven computational model.

QDI circuits offer automatic fine-grained activity gating behavior in the absence of events, reducing power consumption when the circuits are idle. Traditional synchronous systems attempt to solve this problem using various clock-gating schemes, which introduce complexity and require timing margins to ensure clock stability—QDI circuits are naturally free of these requirements.

ULSNAP, is targeted at this sensor mote application space. When idle, our chip consumes only 9 µW with leakage power as the only contributor. It also has fast wake up time, transitioning from idle to active in only 6.5 ns. When active, the 90 nm test chip delivers 93 MHz at 1.2 V and 47 MHz at 0.95 V using 47 pJ and 29 pJ per cycle, respectively. In both the high performance and the low energy mode, ULSNAP is Pareto-optimal in the energy-performance space relative to other state-of-the-art microcontrollers in its class. In fact, ULSNAP can seamlessly operate at different points on the energy-performance curve by scaling its operating voltage.

5.1 Event-Driven Architecture

ULSNAP is a 16-bit architecture and has MIPS-like load/store RISC ISA. The details of the ISA implementation can be found in Appendix B. While ULSNAP's ISA is fairly standard, the execution model of ULSNAP is event-driven. The initial state of the ULSNAP core is a *wait* state. When an event is triggered, e.g. sensor data arrives, ULSNAP's Event Handler (`EH`) references the Event Register Table (`ERT`), which maps each type of event to a program. The `EH` then initiates the fetch of the appropriate instruction stream for the program indicated by the `ERT` and execution begins. Simultaneous events are handled by arbitration within the `EH`. In some cases it is necessary to trigger an event after some delay. We support this functionality with a timer coprocessor, which contains three decrementing counters—allowing us to delay up to three events. At a count of zero, an event is injected into the event queue and is handled by the `EH`/`ERT`. Specifying a delay time is as simple as initializing a counter to the appropriate value.

Each program is terminated by a `WAIT` instruction, returning the processor to the *wait* state. Thus, ULSNAP is in an idle state when no events are available for processing. Note that there is no explicit power- or clock-gating—idle QDI circuits only consume leakage power in the absence of data.

ULSNAP naturally exploits the data-driven nature of QDI circuits: during a quiescent phase, the underlying circuitry simply waits for data to appear. In such an idle state, no switching activity is present and only leakage power is consumed, achieving a low power envelope. No power management controllers, or clock gating techniques are required to support this behavior. In fact, the `WAIT` instruction is for architectural bookkeeping only. Even a stalled program experiences the benefits of QDI circuits. Only the functional units that can make forward progress have

switching activity—inactive or stalled units only consume leakage power. It is important to note that even if the core is in a quiescent state it is ready to compute data—ULSNAP can "wake up" in only 6.5 ns, as detailed in Sec. 5.4.1.

The sequential CHP of the processor core and datapath is shown in Program 5.1. We expand some functions as follows: $get(n) : r\texttt{[}n\texttt{]}$, if $1 \leq n \leq 14$, $get(15) : IN?$, and $get(0) : skip$. Similarly, in Program 5.1, $put(n, v) : r\texttt{[}n\texttt{]} := v$ if $1 \leq n \leq 14$, $get(15, v) := OUT!v$, and $get(0, v) := skip$.

5.1.1 Microarchitecture

ULSNAP and its predecessor, SNAP [12], implement a 16-bit load-store RISC ISA that supports arithmetic, logic, and branching operations. We have a `gcc`-based toolchain that allows us to compile and execute arbitrary C code on ULSNAP. A detailed description of the instructions and its encodings can be found in Appendix B. Instructions are variable length—one to two 16-bit words. A system level diagram of the architecture is shown in Fig. 5.1. ULSNAP has a more streamlined ISA than SNAP, new I/O and timer coprocessors, and an improved memory architecture.

The processor state in ULSNAP is composed of 16 general purpose registers, a PC register, 4 kB of data memory and 4 kB of instruction memory. The `FETCH` unit addresses the instruction memory and forwards instructions to the `PREDECODE` unit, which then resolves the opcode, source, and destination operands from the incoming instruction stream. All fields of the instruction are passed to the `DECODE` unit, which controls operand flow between the register file (`RFILE`) and all execution units. `DECODE` also controls PC update in the `FETCH`, and any required

Program 5.1 Top level CHP of ULSNAP datapath and processing core

$i := 0;\ \ [i < 15 \longrightarrow r[i] := 0; i := i + 1];$
$pc :=$ INIT_PC;
$imask{\downarrow}t$
$*[$
$\quad i := imem[pc];\ \ pc :=\ \ pc + 2$
$\quad [offset(i.A) \longrightarrow offset := imem[pc]; pc := pc + 2$
$\quad [] \neg offset(i.A) \longrightarrow skip$
$\quad]$
$\quad [alur(i) \longrightarrow (v, c) := aluf(get(i.C), get(i.D), i.A, i.D, c); put(i.B, v)$
$\quad [] alui(i) \longrightarrow (v, c) := aluf(get(i.C), offset, i.A, i.D, c); put(i.B, v)$
$\quad [] shift(i) \longrightarrow v := shift(get(i.C), get(i.D)); put(i.B, v)$
$\quad [] shifti(i) \longrightarrow v := shift(get(i.C), i.D); put(i.B, v)$
$\quad [] bfs(i) \longrightarrow put(i.B, bitfieldset(get(i.C), get(i.B), offset))$
$\quad [] loadi(i) \longrightarrow put(i.C, imem[get(i.B) + offset])$
$\quad [] storei(i) \longrightarrow imem[get(i.B) + offset] := get(i.C)$
$\quad [] loadd(i) \longrightarrow\ \ [memreg\ \ =\ \ 0 :\ \ put(i.C, dmem[get(i.B) + offset])$
$\qquad\qquad\qquad [] memreg\ \ =\ \ 1 :\ \ put(i.C,\ \ external\_dmem[get(i.B) + offset])$
$\qquad\qquad\qquad]$
$\quad [] stored(i) \longrightarrow [memreg\ \ =\ \ 0 :\ \ dmem[get(i.B) + offset] := get(i.C)$
$\qquad\qquad\qquad [] memreg\ \ =\ \ 1 :\ \ external\_dmem[get(i.B) + offset] := get(i.C)$
$\qquad\qquad\qquad]$
$\quad [] jal(i) \longrightarrow put(i.B, pc); pc := offset$
$\quad [] jalr(i) \longrightarrow put(i.B, pc); pc := get(i.C)$
$\quad [] beq(i) \longrightarrow [get(i.B) = get(i.C) \longrightarrow pc := pc + offset [] else \longrightarrow skip]$
$\quad [] bne(i) \longrightarrow [get(i.B) = get(i.C) \longrightarrow skip [] else \longrightarrow pc := pc + offset]$
$\quad [] bgez(i) \longrightarrow [get(i.B)\{15\} = 0 \longrightarrow pc := pc + offset [] else \longrightarrow skip]$
$\quad [] bltz(i) \longrightarrow [get(i.B)\{15\} = 1 \longrightarrow pc := pc + offset [] else \longrightarrow skip]$
$\quad [] schedhi(i) \longrightarrow SCHED!get(i.C), TCOP!\mathsf{sched}, TSREG!get(i.B)$
$\quad [] schedlo(i) \longrightarrow SCHED!get(i.C), TCOP!\mathsf{sched}, TSREG!get(i.B)$
$\quad [] cancel(i) \longrightarrow TCOP!\mathsf{cancel}, TSREG!get(i.B)$
$\quad [] ldr(i) \longrightarrow memreg :=\ \ get(i.C)$
$\quad [] rand(i) \longrightarrow put(i.B, \mathtt{lfsr})$
$\quad [] seed(i) \longrightarrow \mathtt{lfsr} := get(i.B)$
$\quad [] wait(i) \longrightarrow EXEC?t; pc := h[t]$
$\quad [] setaddr(i) \longrightarrow h[get(i.C)] := offset$
$\quad]$
$]$

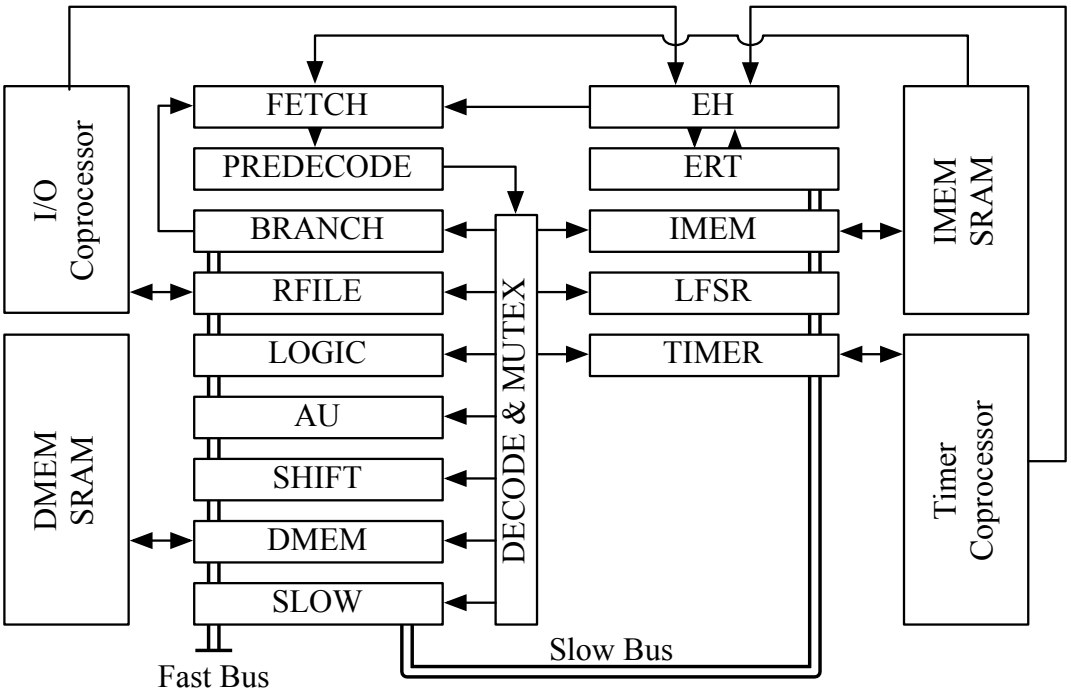


Figure 5.1: ULSNAP Architecture

absolute/relative PC offsets are calculated in the `BRANCH` execution unit.

In designing the overall microprocessor architecture, we leverage the average-case performance properties of QDI circuits (Sec. 2.1) and divide the execution units in ULSNAP into fast and slow groups to improve the overall energy efficiency and performance. Operands and results are transported between execution units and the register file (`RFILE`) by four shared buses: X and Y for register source operands, Z for immediate values, and W for results. Frequently used execution units (`RFILE`, `JUMP`, `BRANCH`, `LOGIC`, `ARITH`, `SHIFT`, `DMEM`) are connected directly to these operand and result buses. Less-critical units, i.e. the `LFSR`, `ISTORE`, `TIMER`, and `ERT` units, are decoupled from the buses by a single, dedicated access unit (`SLOW`) as shown in Fig. 5.1.

This effectively creates two sets of operand and result buses, logically and elec-

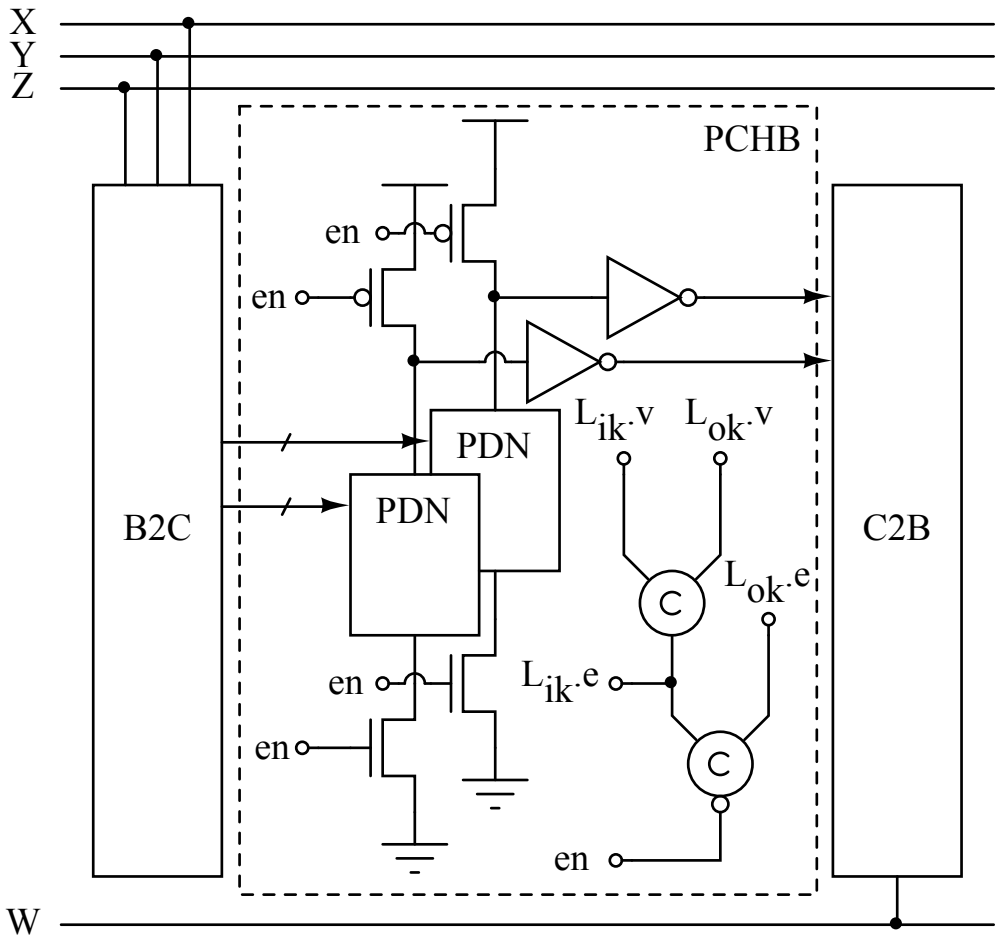
trically separating the execution units into fast and slow groups. This has the benefit of significantly shortening the bus wires and reducing per-bus capacitance. For example, we estimate that a monolithic X bus would have in excess of 0.4 pF/wire of total capacitance, accounting for both coupling and intrinsic capacitance. Instead the capacitance is split in two segments of 0.17 pF/wire and 0.2 pF/wire for the slow and fast buses respectively. Access to the slow buses incurs an extra overhead of 2 gate delays and an intermediary 0.1 pF/wire.

While the total system capacitance is greater than the estimated monolithic bus capacitance, most of the time the slow units are not accessed. Most operations use only the fast units and therefore only see 0.2 pF. This increased performance for common operations offsets the added latency of access to the slow units and improves overall system performance as we are improving the *average* case execution paths. The non-uniform run times for the execution units poses no synchronization problem since our self-timed methodology is robust to gate and wire delays (Sec. 2.1). We quantify the relative difference between the slow and fast execution paths using a synthetic benchmark, discussed in Sec. 5.4.1.

5.1.2 Circuit Implementation

We make use of a hybrid approach at the circuit level, combining two different QDI logic families: precharge buffer templates [33] and control-data decomposition. These two families fall at opposite ends of the spectrum of pipeline stage complexity. Precharge buffer pipeline templates, such as PCHB and PCFB, were widely used in the MiniMIPS processor [38]. Each PCHB/PCFB stage typically implements a function of small enough complexity that it can fit into a single nMOS pulldown network, as illustrated in Fig. 5.2. This compilation style yields high-

performance stages with short cycle time. However, a reasonably complex function must be decomposed into a pipeline of several PCHB/PCFB stages, resulting in a long latency for a single data token to travel through the entire pipeline, though maintaining a high token throughput.



Center shows bit slice of the datapath an of execution unit.

Figure 5.2: ULSNAP Execution Unit Template

Conversely, control-data decomposition, used in the Caltech Asynchronous Microprocessor [41], typically aggregates computation into a single pipeline stage. While the cycle time of such a stage is higher than the equivalent PCHB/PCFB pipeline, the overall latency and energy consumption are less. Circuits compiled using either of these methods are completely inter-operable, which allows the de-

signer to tailor the latency/cycle time of all computational units individually.

We implemented high throughput execution units such as the ARITH and BRANCH units using the PCHB/PCFB templates. Fig. 5.2 depicts an example bit slice of such an execution unit. Given the small amount of computation these units perform, the PCHB/PCFB pipeline is at most 2 stages. This represented a reasonable tradeoff between throughput and energy/latency for these stages. The fetch loop and predecode units are compiled using the control-data technique, which allowed us to minimize the latency of key computations such as updating the PC.

As described earlier, all the functional units are connected by operand (X, Y, and Z) and result (W) buses, each of which is a shared channel. Channels only provide synchronization between a single produce and consumer, i.e. they are not multicast. Some additional hardware is necessary to preserve the local synchronization handshakes described in Sec. 2.1, so we wrapped each unit with bus-to-channel (B2C) and channel-to-bus (C2B) interfaces, which are controlled by the DECODE unit. As an example, Prog. 5.2 the B2C interface for the X bus and the C2B interface for the W bus in CHP[1]:

Program 5.2 Bus-to-channel and channel-to-bus programs

$$
\begin{aligned}
&B2C \;\equiv \\
&\quad *[[\overline{Read_k}]; \; x_k\uparrow; \; Read_k; \; L_{ik}!(X?); \; x_k\downarrow] \\
&C2B \;\equiv \\
&\quad *[[\overline{Write_k}]; \; x_k\uparrow; \; Write_k; \; W!(L_{ok}?); \; x_k\downarrow]
\end{aligned}
$$

Fig. 5.2 shows a PCHB-style functional unit with B2C and C2B interfaces. The internal PCHB pipeline stage accepts input on channel L_{ik} and produces outputs on channel L_{ok}. The $Read_k$ and $Write_k$ are dataless channels connecting the DECODE unit to each of the B2C and C2B interfaces. We have expanded the above

[1]A short summary of CHP can be found in the Appendix

CHP descriptions to include an internal state variable x_k, which we discuss later. By interfacing with the appropriate $Read_k$ or $Write_k$ channel(s), the decode unit can guarantee each functional unit mutually exclusive access to the appropriate operand and result buses.

Unlike all other functional units, the `DECODE` unit is not implemented with PCHB/PCFB or control-data style pipeline templates, as it must provide resource allocation functionality. To address this specific need we make use of Pipelined Mutual Exclusion (PME) [36]. In short, by using PME the `DECODE` unit synchronizes the fetch and execution units while allowing the fetch loop to continue execution as the execution unit(s) processes the previous instruction. This simple optimization allows us to introduce concurrency with little overhead.

PME can be described as follows: given a set of mutually exclusive actions $(A_1, A_2, \ldots, A_n)$ and a command channel to execute each of those actions $(C_1, C_2, \ldots, C_n)$ we can guarantee mutual exclusion by Program 5.3.

Program 5.3 Pipelined mutual exclusion

$$
\begin{array}{l}
p_i \equiv \\
\quad *[[\overline{S_i}]; \;\; x_i\uparrow; \;\; S_i; \;\; A_i; \;\; x_i\downarrow] \;\; \| \\
\quad *[[\overline{C_i} \wedge (\forall j : j \neq i : \neg x_j)]; \;\; S_i; \;\; C_i]
\end{array}
$$

Note that the above CHP program consists of two separate programs running in parallel, synchronized by synchronization channels $S_1, S_2, \ldots, S_n$. The first process of p_i is structurally identical to that of the `B2C` and `C2B` processes, which essentially replace A_i with the appropriate channel actions on the bus and local channels. In the PME context, the set of state variables $x_0, x_1, ..., x_n$ behave as a distributed synchronization lock that reserves resources when executing an action, e.g. the shared bus channels.

The key feature of PME is how it allows a control process communicating on the command channels C_j, in this case the decode logic, to continue execution without waiting for the commanded action A_j to complete. To illustrate this, let us assume that the action A_k is desired, and that it is the first action, i.e. there are no current actions being performed. The controlling process initiates a channel action on C_k. The wait condition $[\overline{C_k} \wedge (\forall j : j \neq k : \neg x_j)]$ is met, so a channel action on S_k is initiated. The next wait condition $[\overline{S_k}]$ is now met, reserving the shared resource by raising x_k. At this time, the channel actions on S_k and C_k are allowed to complete, freeing the controlling process to continue work.

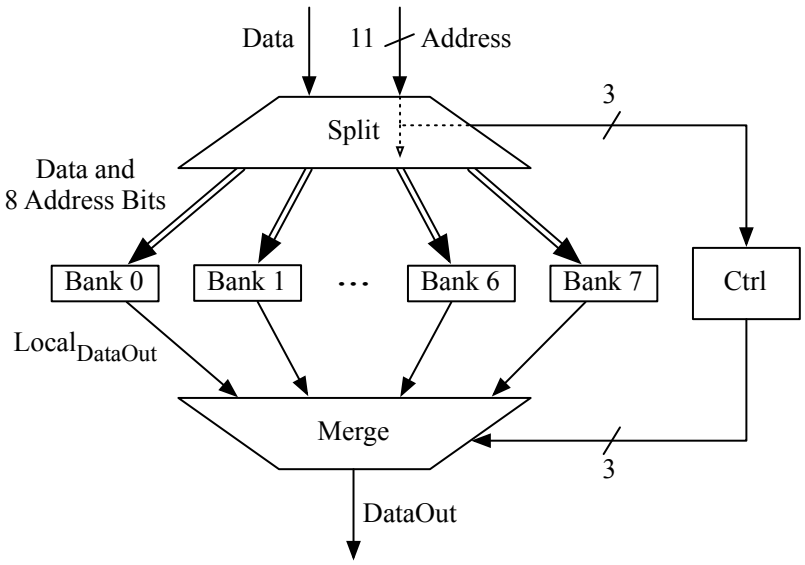
Synchronization happens when the `DECODE` unit tries to execute the next action by initiating the appropriate command channel action on C_i. At this point, the controller must wait for all locks x_j to become **false**, which in our example will only occur once A_k has finished. By decoupling the `DECODE` unit from function unit action completions, we can begin decoding an instruction while the previous instruction is being executed, adding additional concurrency to our execution.

5.2 Memory Architecture

ULSNAP implements a Harvard memory architecture, with a different set of address and data buses for instruction memory and data memory. Data width and instruction width are both 16-bits, hence both memories are 16-bit aligned.

ULSNAP has 8 kB of memory, divided equally into instruction and data memory. The symmetric size of the memories was just for convenience, since they do not share the same address space. Both memories are organized into 8 banks as shown in Fig 5.3. A memory operation is handled by a `SPLIT` process that addresses the

correct bank using the least significant bits of the address. The `SPLIT` process was compiled using a full buffer reshuffling (PCFB) that allows multiple outstanding operations—up to one per bank. Read operations make use of the `MERGE` process, which selects the appropriate $\text{Local}_{\text{DataOut}}$ bus and ships the result back to the core. In the case of contiguous memory access or small strides, we can leverage our ability to have multiple outstanding memory operations to different banks. In this way the PCFB reshuffling enables us to reduce performance requirements for each bank without starving the processor core.



Banks are addressed using LSB bits of the address. Only active banks consume dynamic power during a memory request.

Figure 5.3: SRAM Organization

In order to further reduce static power, the SRAM bit cell relies on long channel devices to reduce leakage. The total SRAM leakage is 4 µW for all 8 kB of memory. For reference, a direct port of the original 180 nm SNAP memory to our more modern 90 nm process consumes more than 200 µW of leakage power. Note that this reduction of static power between designs does not come at the cost of a

significant latency increase. In fact, due to the multiple outstanding requests enabled by our use of PCFB-reshuffled logic, the average SRAM access latency of ULSNAP is similar to that of a single cycle of the microcontroller core.

Each bank is divided into 64 rows and 4 columns, each of which is 4 B (2 words) wide. We chose this configuration to allow for relatively short bit lines. Shortening the bit lines reduces switching capacitance and improves noise margins. Reads from the SRAM are fully QDI, since a read operation will eventually cause a bit line transition which can be detected. However, we cannot observe the state of a write operation by only inspecting the bit lines. In order to provide a timing bound for a write operation, we build a delay-line-like structure out of a dummy SRAM column, placed on the side of the SRAM farthest from the word line drivers. During a write operation on the SRAM, we perform a read on this dummy column and wait for its transition to be detected.

The placement of the dummy column at the end of the word line accounts for the maximum possible delay on the word line. Furthermore, since this dummy column is identical in every other respect to actual SRAM columns, the bit line capacitance charge/discharge timing characteristics are comparable. The key assumption is that reads take longer than writes, padding our delay margins. This configuration allows us to have a dummy delay-replica-loop that approximates the delays associated with the physical design as well as global and systemic process variations.

5.3 Coprocessors

5.3.1 Timer Coprocessor

Providing efficient hardware support to schedule events in the future is crucial in order to maximize the amount of time the ULSNAP core can remain idle, using only leakage power. To this end, both SNAP and ULSNAP implement timer coprocessors. The timer coprocessor in ULSNAP is composed of three 24-bit decrementing counters or "timers." Each counter can be independently initialized to a positive integer through the use of two custom instructions in the ULSNAP ISA: `SCHEDHI` and `SCHEDLO`. These instructions set the most and least significant bits of each counter, respectively. When a timer expires, i.e. the counter has been decremented to 0 from the initial value, the timer coprocessor injects an event into the Event Handler (`EH`) event queue.

The original SNAP timer coprocessor was constructed from a single always-running clock or "tick generator" and three decrementing counters. Gating the clock signal connection to each of the counters enabled or disabled each of the counters, providing three controllable timers. While simple to implement, this approach did not leverage a key benefit of asynchronous circuits: intrinsic activity gating. The use of a continuously running clock is power inefficient, especially when considering the required distribution to three counters.

ULSNAP makes an improvement to this design by implementing per-timer, stoppable clocks for each one of the three counters. This enables per-timer activity gating and reduces the amount of global wiring. Furthermore, the frequency of each one of the clocks is configurable, allowing for different wait times and wait time

precision. Each of the timers is completely decoupled from the others providing significant savings in power consumption. Program 5.4 shows the sequential CHP for the timer coprocessor. Note that the program must issue a `SCHEDHI` instruction before an `SCHEDLO` instruction, otherwise the timer may deadlock. In order to avoid a race condition when the core cancels a timer that just expired, the timer inserts a token into the execution queue whenever a `CANCEL` operation is processed.

Program 5.4 CHP for the timer coprocessor

```
*[ TCOP?op;
   [] op = schedhi  -->  SCHED?temp[16:0], TSREG?
   [  op = schedlo  -->  SCHED?temp[15:0], TSREG?t;
                         enable[t]; SYNC[t]!temp;
   [] op = cancel   -->  TSREG?t; CANCEL[i]!
   ]
 ]

<|| : i : 3 :

s[i] := off
*[ [~CANCEL[i] ∧ s[i] = off   -->  CANCEL[i]?
   | ~CANCEL[i] ∧ s[i] = on    -->  CANCEL[i], EXEC_t[i]!, s[i] := off
   | ~DEC[i] ∧ s[i] = on       -->  timer[i] = timer[i] - 1; DEC[i]?
                                    [timer[i] = 0 --> EXEC_t[i]!, s[i] := off
                                    [] timer[i] > 0 --> DECREMENT[i]!
                                    ]
   | ~SYNC[i]                  -->  SYNC[i]?(timer[i]); s[i] := on;
                                    DECREMENT[i]!
   ]
 ]
||
Tick_Gen[i]  ≡
*[ [DECREMENT[i]?; DEC[i]! ]

>
```

A detailed picture of each timer can be seen in Fig. 5.4. Each timer has a tick generator (Tic Gen) that generates tokens on a dataless asynchronous channel (*dec*) at a user-configurable frequency. The *dec* channel serves as the command to decrement the counter. The ULSNAP core configures, sets, starts, stops, and resets each timer via the *ctrl* channel. As timer commands can arrive even if the

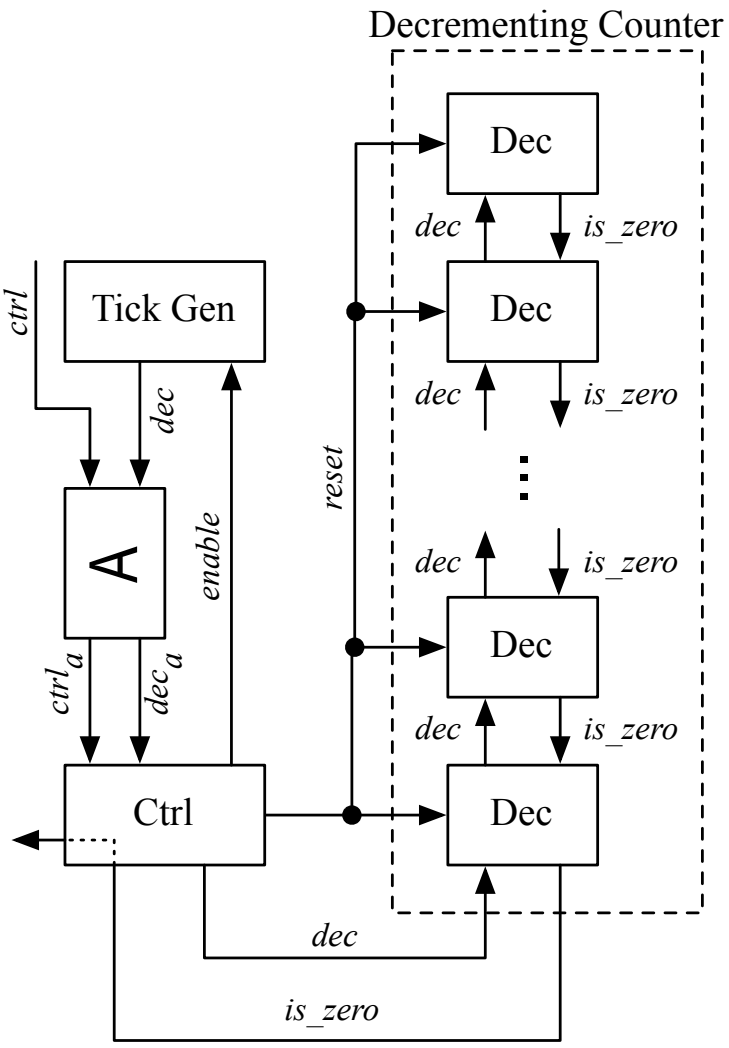


Figure 5.4: Timer Implementation

timer is active, especially if the timer is being reconfigured or reset, the *ctrl* and *dec* channels must be arbitrated. The resulting $ctrl_a$ and dec_a channels are mutually exclusive. The Ctrl process is responsible for initializing the decrementing counter, enabling/disabling the Tic Gen process, as well as detecting when the counter is zero and injecting an event into the Event Handler.

The decrement counter Fig. 5.4 is implemented as a serial pipeline of n-bit decrementer processes (Dec), each of of which corresponds to a single bit of counter.

The Ctrl process can reset and initialize each unit via a *reset* channel, which is connected to each decrementer process. Each Dec process stores two variables: b, the actual value of the counter bit, and z, a bookkeeping value which is true if all bits more significant than the current position are zero. The z value enables two key features. The first is to minimize the number of exercised Dec units while decrementing—if all higher order bits are zero, there is no need to interact with them thus saving power. Secondly, because each Dec unit has information about itself as well the higher bits relative to itself, the decrement action is constant response time, similar to the empty pipeline detection counter of [55]. If a decrement operation must borrow from a more significant bit, a channel action takes place on the *dec* channel. The next Dec process responds with the appropriate z value on the is_zer_o channel to keep all the state updated. As there is no clock, a constant-time asynchronous cycle is of great importance in a decrementing counter used as a timer.

5.3.2 I/O coprocessor

Off-chip communication is handled by an I/O coprocessor. In comparison to SNAP, ULSNAP's I/O coprocessor has been made more modular, enabling easy support of different serial protocols. Currently, we implement two off-chip serial protocols in the I/O coprocessor: SPI, and a simple asynchronous serial protocol similar to that shown in Fig. 2.1. Communication between the I/O coprocessor and the core is done through an I/O-mapped register (`R15`). Whenever an I/O event occurs, an token is placed into the Event Handler queue and the associated data is pushed into the input queue of `R15`.

The SPI unit can only be used in master mode. The frequency of the SPI clock

can be configured through an off-chip delay line. We support all SPI clock polarity (`CPOL`) and phase (`CPHA`) modes, each of which can be configured by initializing the `SPICFG` register in the I/O coprocessor to the appropriate values. In order to avoid a race condition between writing to the `SPICFG` register and the transmit (`SPITX`) or receive (`SPIRX`) registers, the I/O coprocessor inserts an event into the Event Handler's event queue whenever the configuration is changed. The SPI unit can be configured in transmit, receive or duplex mode. To preserve the event-driven architectural model, the I/O coprocessor will inject an event into the Event Handler queue whenever a word is received through the SPI or serial unit. The throughput of the serial asynchronous interface of the I/O coprocessor is a 16 bit serial message every 2 cycles (130 ns). The throughput of the serial I/O interface is limited by our padframe design and the capacitance of the PCB traces.

5.4 Evaluation

5.4.1 Testchip

ULSNAP is the successor to the SNAP processor and is fabricated in a more modern 90 nm low-power CMOS process using a full-custom layout flow. When appropriate, we make the relevant comparisons between a simulated 90 nm ULSNAP and our ULSNAP design. The processor core alone contains 122k transistors in an area of 0.312 mm^2. Including the memories, the transistor count is 592k in an area of 0.844 mm^2. *All* reported power measurements include the memory power consumption. A photo of the die is shown in Fig. 5.5. Our test chip included 3 designs, ULSNAP is located on the southeast quadrant of the test chip.

A picture of the layout is shown in Fig. 5.6. The layout is roughly divided in 3 rows. On the top row we placed the `FETCH`, `RFILE`, and the *fast* group of the execution units. On the middle row, we placed the `DECODE` unit. The bottom row includes the I/O and timer co-processors as well as the *slow* group of the execution units. The X, Y, Z, and W buses run horizontally and "tap" each one of the fast group of execution units and the `RFILE`. The buses of the *slow* execution units connect to the to the main buses through a vertical bus.

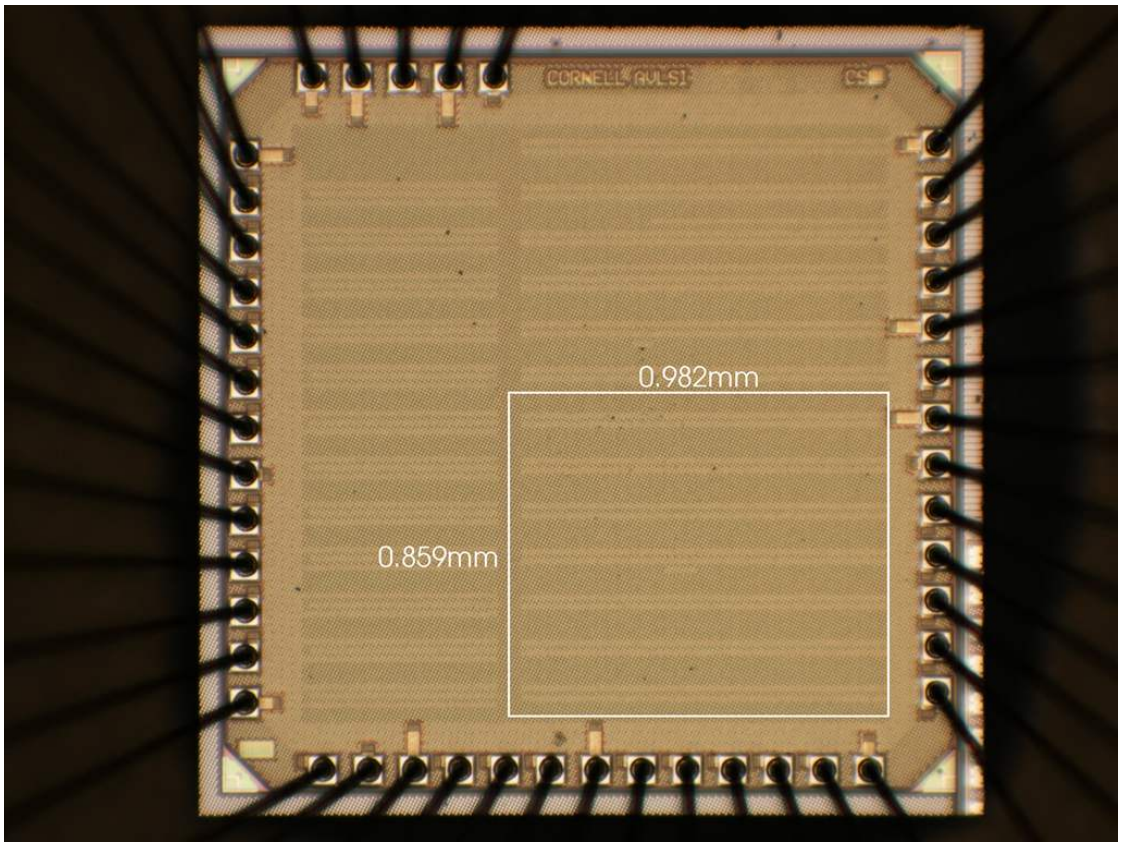


Figure 5.5: Die photo

Figure 5.7a shows a picture of the PCB board used to program and evaluate the test chip. The board can be connected to an Arduino Mega as a shield. The board has several modes of operation, on one hand, it can be connected to a high precision bench equipment to perform measurements and analysis. Alternatively, it can operate on stand-alone mode, where internal sensors work together with an

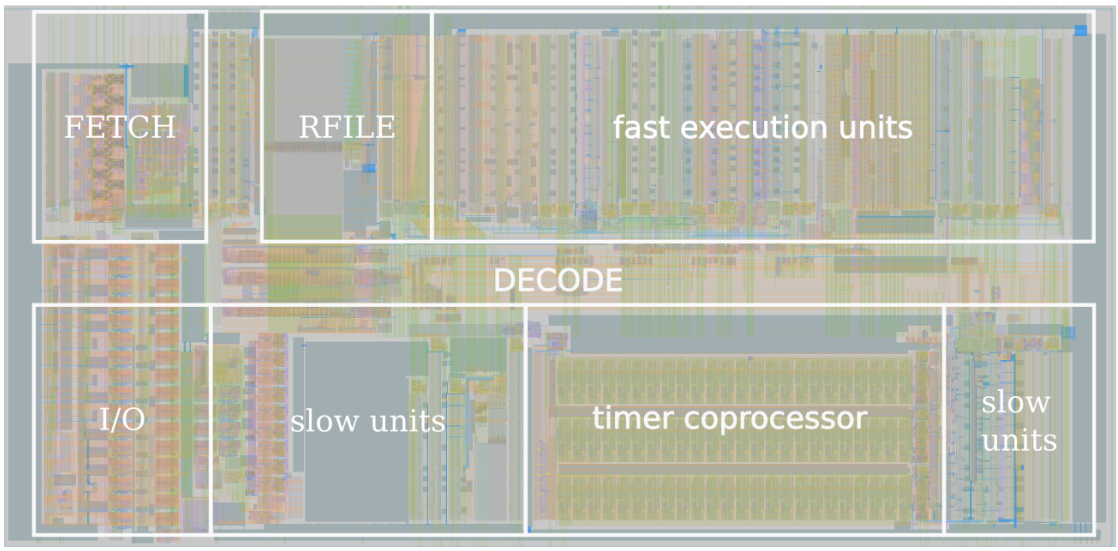
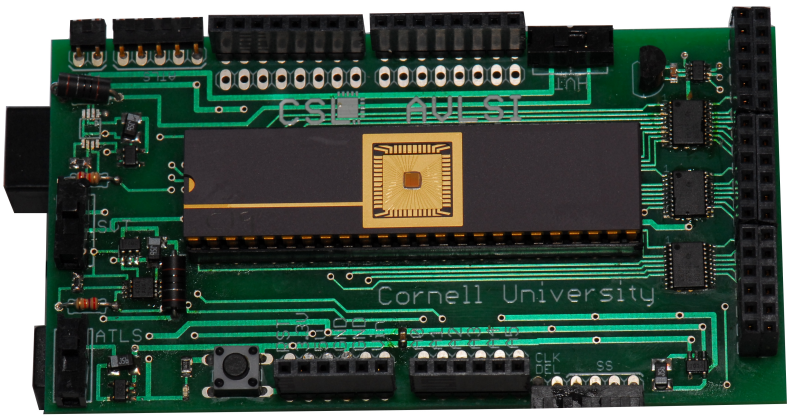


Figure 5.6: Layout Photo of core without memory

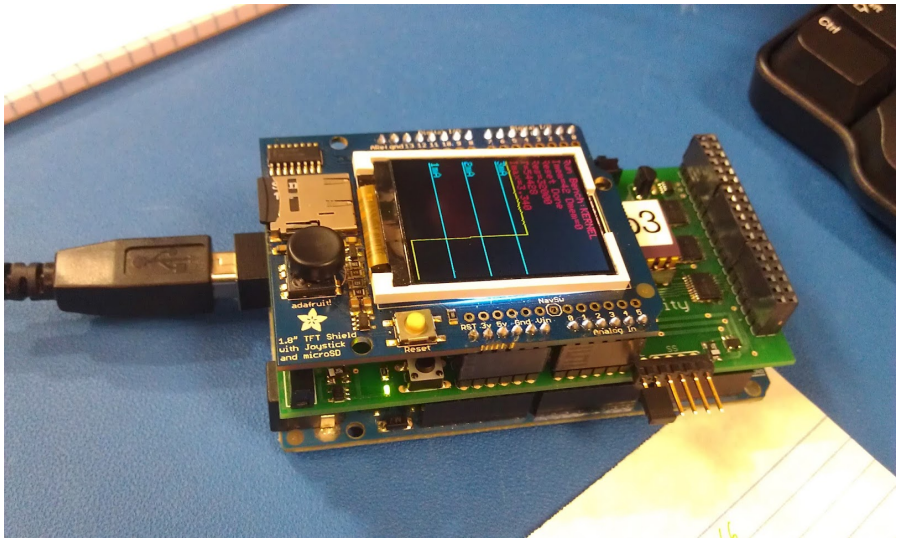
Arduino Mega board in order to bootstrap and benchmark applications running on the board. Figure 5.7b shows the PCB connected as an Arduino shield operating in stand-alone mode. In this mode, the USB connection only provides power to the board, all the benchmarks are loaded on an SD card, and the results are displayed on the LCD. More details of the board are in Appendix C.

5.4.2 Energy, Throughput, and Lifetime

To evaluate the static power consumption, we tested 10 separate ULSNAP chips with empty event queues, i.e. there was no activity. Since QDI circuits provide automatic fine-grained activity gating, and our design does not contain any busy loops or waits in the absence of events or instructions, leakage power is the only source of power consumption while idle. Note that this is not an explicit power-gated state and that there are no explicit hardware power management structures in ULSNAP—although extending the design to include them is possible as described in Chapter 4. Fig. 5.8 presents measured static power consumption from



(a) PCB photo



(b) Portable Test Board

Figure 5.7: Test, simulation, and measurement board

all leakage paths, and also shows how static power scales with V_{DD}.

To evaluate the processing performance of ULSNAP when active, we developed micro-benchmarks that stress the processor with ALU and memory operations. We measured the performance and energy for our micro-benchmarks while varying the supply voltage from 1.2 V to 0.95 V. As ULSNAP has no clock, we have no direct

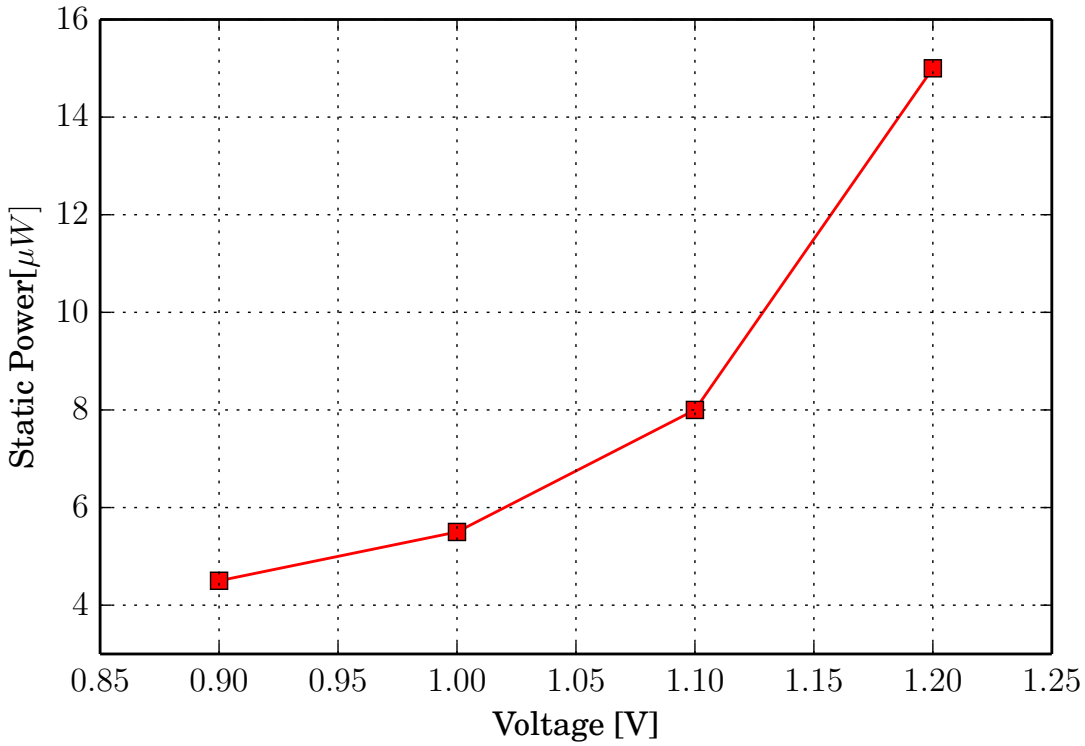


Figure 5.8: Static power consumption

control over the operating frequency save for changing V_{DD}. Note that this voltage and "frequency" scaling is a natural benefit to our use of QDI circuits and requires no explicit hardware support or design effort.

We report power and performance characteristics in Fig. 5.9. Our 90 nm test chip delivers an average of 93 MHz at 1.2 V using 47 pJ per cycle. These numbers are an average of measurements across 17 different ULSNAP cores with standard deviation 5 MHz and 0.5 pJ. ULSNAP can be also run in low energy mode with V_{DD}at 0.95 V. In this mode, it only uses 29 pJ per cycle while delivering 47 MHz of integer operations.

Fig. 5.9 also shows that ULSNAP automatically adjusts to multiple voltages, allowing it to operate at different points on the energy-performance curve. A

smart application could control the supply voltage to ULSNAP using a digitally controlled power source/regulator to optimize the energy-performance trade off during the lifetime of the sensor mote. While this approach is possible on synchronous circuits, it requires a focused design effort to close timing.

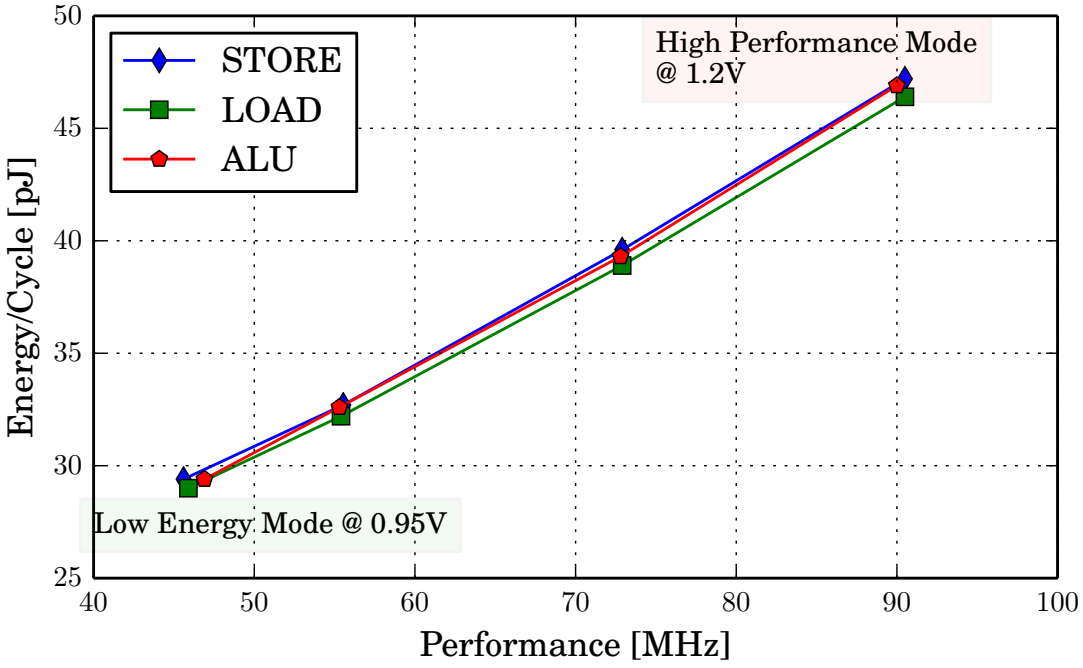


Figure 5.9: Performance-energy trade off

We also include an evaluation of six standard benchmarks for embedded processors [47] implemented in C and compiled to our custom ISA with our `gcc` flow. Performance and energy measurements at 1.2 V for each benchmark are shown in Table 5.1.

Finally, we developed a micro-benchmark to stress the timer coprocessor with the main core idle—this is possible as the three timers are decoupled from the main core. Our measurements show that each timer can reach average frequencies up to 270 MHz while consuming only 0.85 pJ/cycle/timer. While each timer's individual frequency is configurable, in our testing we ran them all at the same frequency.

When idle, ULSNAP's timer coprocessor uses only 300 nW at nominal V_{DD}, as compared to 400 µW for the coprocessor in [12].

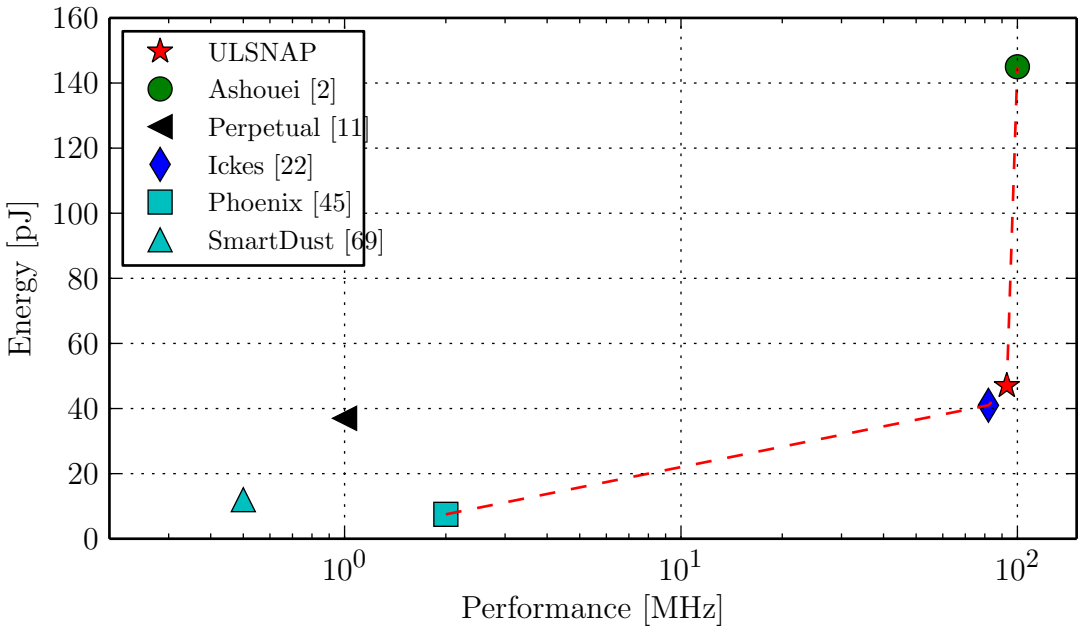


Figure 5.10: Energy-performance comparison of processors in high performance mode. The dotted line connects the microcontrollers on the Pareto-optimal set

We compared ULSNAP against various state-of-the-art microcontrollers [2,11,22,45,69] and present the results in Table 5.2. Fig. 5.10 and Fig. 5.11 shows the comparison of the microcontrollers on the energy-performance design space. The Pareto-optimal set is highlighted by a dotted red line. ULSNAP is Pareto-optimal in the energy-performance space in both low-energy and high performance

Table 5.1: Benchmarks

Task	Perf [tasks/s]	E [nJ/task]	Input
CRC4	3.19×10^5	12	16b
Tiny Encryption (TEA)	8.41×10^3	490	64b(data)
Int Average	6.37×10^3	652	2kB
MinMax	4.73×10^3	821	2kB
Search	1.55×10^3	27	2kB
Serial RX	1.63×10^3	7	16b
AES-Encryption	14.4×10^3	283	128b

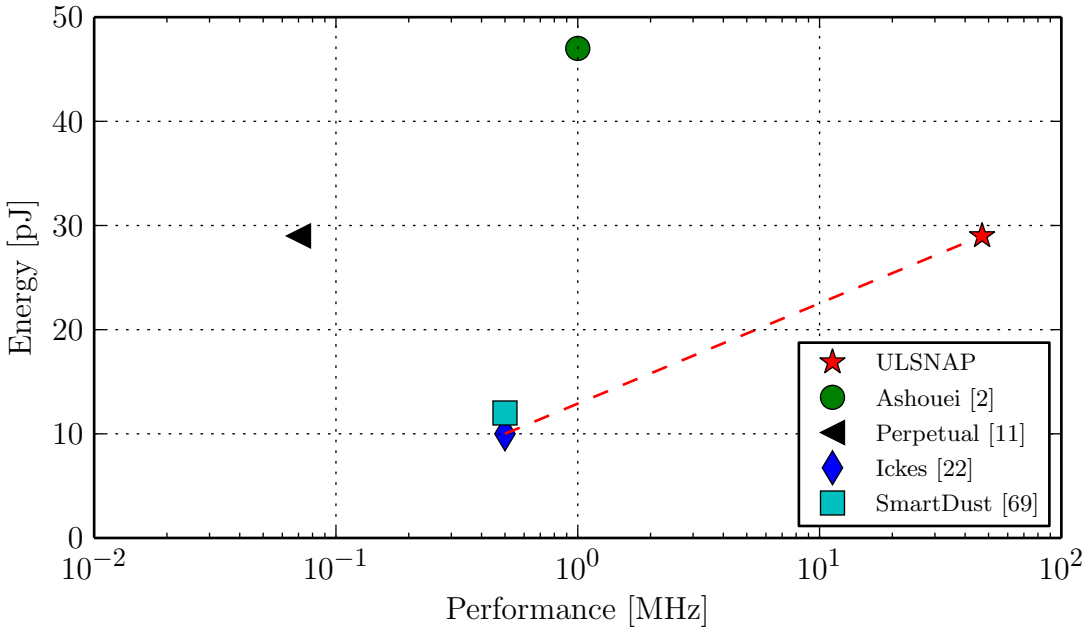


Figure 5.11: Energy-performance comparison of processors in low energy mode. The dotted line connects the micocontrollers on the Pareto-optimal set

Table 5.2: Comparison of State of the Art Microcontrollers

	[2]	[11]	[22]	[45]	[69]	ULSNAP
Tech [nm]	90	180	65	180	250	90
Datapath [bits]	24	32	32	8	8	16
SRAM [kB]	2000	3	16	0.33	3.12	8
High Performance Mode						
Supply [V]	1.2	0.5	1.2	0.9	1.0	1.2
Perf. [MHz]	100	1	82	2	0.5	93
Energy [pJ]	145	37	41	7.5	12	47
Low Energy Mode						
Supply [V]	0.4	0.4	0.5	0.5	NA	0.95
Perf. [MHz]	1	0.07	0.5	0.1	NA	47
Energy [pJ]	47	29	10	2.8	NA	29

Reported numbers for High Performance Mode are for minimum cycle time workloads. Low Energy mode numbers are for workloads which minimize energy.

modes relative to all other state-of-the-art microcontrollers. In other words, ULSNAP is superior in either performance or energy, if not both metrics, as compared to other deeply embedded microcontrollers. This is achieved by a combination of factors: ULSNAP's event-driven design, micro-architectural optimizations such as

bus partitioning, and circuit implementation details such as the use of self-timed circuits. As power and performance numbers are workload dependent, the numbers reported in Table 5.2 and Fig 5.10 are for the workload that performs best on each microcontroller.

We also evaluated the node lifetime of a theoretical mote built with the different processors in Table 5.2 using the proposed model discussed in Chapter 3. We assume that we use the TI-CC1101 transceiver for RF communication, which offers 500kps of bandwidth and uses 55 mW. We also assume that all processors can be paired with the CC1101 or an equivalent radio.

To level the playing field as much as possible, we also paired each microcontroller with a Power Management Unit (PMU). ULSNAP's PMU consists on a voltage scalable switched capacitor similar to the one in [60]. This DC-DC converter has efficiencies greater than 75 % across a wide range of loads. We assume that microcontrollers [45,69] are paired with PMU consisting of a Fibonacci switched capacitor network and a low-dropout linear regulator [71]. The Fibonacci-based PMU has efficiencies of 62 % on active mode and 12 % when drawing less than 1 µW of power. An exception to the previous assumption is the microcontroller presented in [11] since it has an integrated Power Management Unit (PMU) that contains a linear regulator with high efficiencies and optimized for active and idle modes. The MSP430 microcontroller has an integrated PMU. While the efficiencies of the MSP's PMU are unknown, our power measurements inherently account for any inefficiencies that DC-DC conversion might offer.

We assumed sensor events arrive with a Poisson distribution and with an average inter-arrival time $1/\lambda$. Each event has a probability α of causing the transmission of a single encrypted packet. The total packet size is 1064 bit with a payload

of 127 B. These payloads are consistent with the payload of an IEEE 802.15.4 (Zigbee/Xbee) standard. The transmission time of a packet, T_T, is set by the transmission rate. Transmitting a 1064 B packet takes $T_T = 2.1$ ms. We assume that a processor operates at maximum performance while active and transitions into a low-voltage sleep mode during idle time.

Since most of the static power consuption of a microcontroller comes from the memories, for fairness, we normalized all microcontrollers to have 8 kB of memory. Table 5.3 show the reported static power consumption by the microcontrollers of Table 5.2, as well as the static power consumption per bitcell. Some publications do not report static power consumption at all, so we were unable to compute the lifetime of the some microcontrollers [2,22].

Table 5.3: Static power consumption of memory blocks and memory bitcells

Processor	Static Power [W]	Memory [kB]	Static Power/cell [W]
SmartDust [58]	13n	3.125	10f
Phoenix [45]	36p	0.33	10f/4f‡
Perpetual [11]	460p	2SRAM + 3R-SRAM†	

†: R-SRAM: Retentive SRAM

‡: Power-gated/not-power gated

We present lifetime as a function of the event arrival date in Fig. 5.12a. We assume an average processing time for each event to be $\bar{\mu_2} = 1.5$ ms in ULSNAP microcontroller, which corresponds to the time it takes to run the statistical benchmark set from SenseBench and encrypt a 127 B payload using a 128-bit AES block cipher on ULSNAP. All performance numbers are taken from the measured results shown in Table 5.1. To estimate the CPU time, $\bar{\mu_2}$, for other microcontrollers, we just multiply by the frequency difference rate between the other microcontrollers and ULSNAP. Encrypting a 127 B payload requires running a 128-bit AES cipher block 7 times, which we have accounted in our evaluation. Table 5.4 presents a

summary of the parameters used for the lifetime model evaluation and Table 5.4b present the empirical values used for the lifetime evaluation.

Table 5.4: Parameters used for node lifetime evaluation

(a) Components of time and power spent on each state

State	Average time spent before transition	Power
S_1 (Idle)	$\frac{1}{\lambda}$	$\frac{P_{off\_peripheral}+P_{off_{\mu}P}}{efficiency_{pmu}}$
S_2 (Process)	$T_{ADC} + WUT_{\mu P} + T_{ULSNAP} \cdot \frac{freq_{\mu P}}{freq_{ULSNAP}}$	$\frac{P_{on\_\mu P}+P_{off\_peripheral}}{efficiency_{pmu}}$
S_3 (TX)	$WUT_{Radio} + T_{Radio}$	$\frac{P_{on\_\mu P}+P_{Radio}}{efficiency_{pmu}}$

(b) Default parameters

Parameter	Description	Default Value
T_{Tx}	Transmission time of 1024bits	2.368 ms
P_{Radio}	Radio TX power	55 mW
WUT_{Radio}	Radio wake-up time	240 µs
T_{ADC}	Time for single ADC conversion	650 ns
P_{ADC}	Power of ADC in active mode	1.047 mW

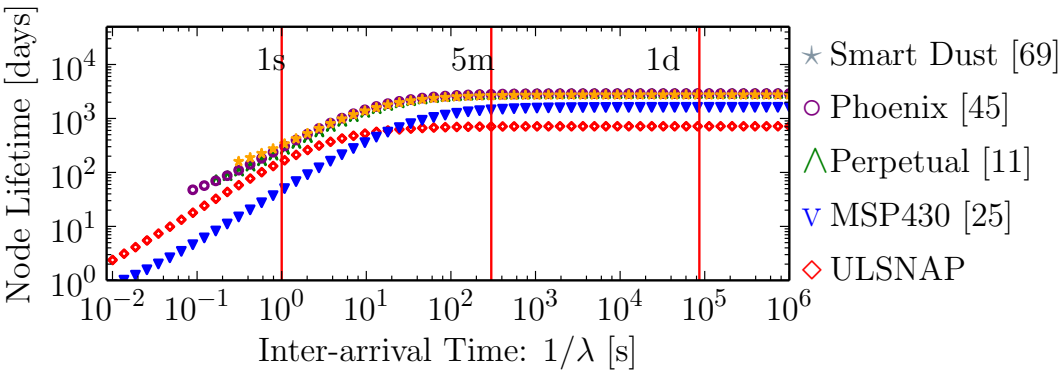
The lifetime of the processors when the workload for each processor, Y is 1 ms is shown in Fig 5.12a. ULSNAP asymptotically approaches a maximum lifetime of 2.1 years when events arrive at rate greater than one event every 10 min. Comparatively, at this same event rate, the processors in [11,45,69] asymptotically approach lifetimes from 7.3 years to 7.7 years. The MSP430 microcontroller has a lifetime of 3.3 years when events arrive at rates $1/\lambda > 10$ min, but its lifetime is affected with great degree as events arrive faster than 1 event per second.

Fig 5.12b shows that increasing the processor load to 10 ms per event does not have a great impact on the node lifetime when events arrive very sporadically. ULSNAP's lifetime is reduced to 1.9 years when events arrive at a rate of 1 event every 10 min. Comparatively, the lifetime of the lifetime of the microcontrollers in [11,45,69] have lifetimes that range from 6.5 years to 7.5 years. In contrast, when the CPU load is higher and event interarrival times are 10 s, ULSNAP mi-

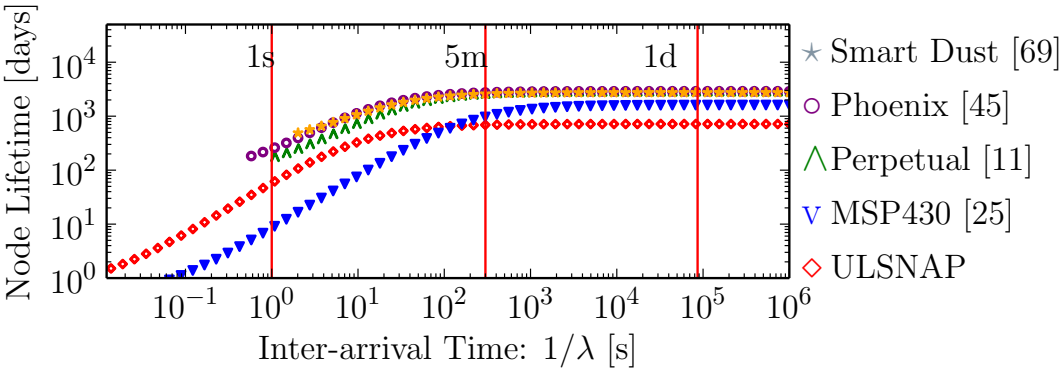
crocontroller has a lifetime of 1 year, while other microcontrollers have lifetimes that range from 2.0 years to 3 years. The gap in lifetimes between ULSNAP and other microcontrollers has been reduced due to ULSNAP's competitive energy usage in high performance mode. While the MSP430 is able to meet deadlines even for fast inter-arrival event rates of 1 event every 10 s, its lifetime is only 0.2 years.

The impact of performance can also be noticed in Fig. 5.12. While ULSNAP's lifetime has a long left tail, other microcontrollers do not enough processing power to handle fast interarrival events, i.e. processing events at fast interarrival rates *unfeasible*. For instance, in Fig. 5.12b, while ULSNAP and the MSP430 microcontroller can process events arriving faster than $1/10^{th}$ of a second, the other microcontrollers can only handle events arriving at roughly 1 event per second. If the CPU workload is increased, as shown in Fig. 5.12c, we see that ULSNAP can handle almost 10 events per second, while the MSP430 can only handle events arriving once per second. In contrast, all the other microcontrollers can only handle 1 event every 10 seconds. A 10 Hz event arrival rate may seem quite high for a WSN application, but the fact that ULSNAP can handle such event rates provides the system designer with additional data processing capabilities. Some examples include calculating a moving average, various filtering operations, or even somehow coordinating computation with other nodes in the WSN.

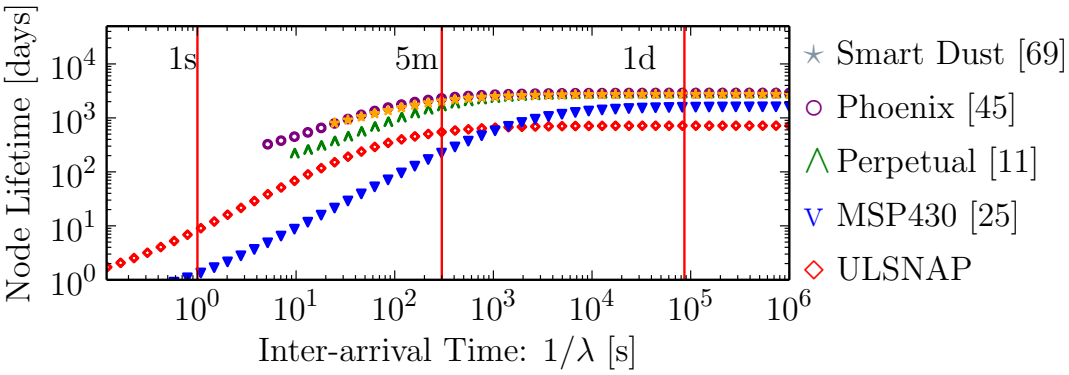
To measure the effect of splitting the datapath buses into two fast and slow buses, we performed an experiment that performs 20×10^6 consecutive memory *writes* first to the Data memory `DMEM` and then to the Instruction memory `ISTORE`. The unit responsible to access `DMEM` is connected to the fast buses while the unit responsible to write into the Instruction memory is connected to the slow buses as shown in Fig 5.1. While this is not an exhaustive test of every functional unit



(a) $T_{\bar{ULSNAP}} = 1\,\text{ms}$



(b) $T_{\bar{ULSNAP}} = 10\,\text{ms}$



(c) $T_{\bar{ULSNAP}} = 100\,\text{ms}$

Figure 5.12: Lifetime comparison between motes between ULSNAP and other microcontrollers as function of inter-arrival time. We assume that we are using a 35 mAh, 3 V CR1220 coin cell battery

on both buses, the DMEM and ISTORE interfaces are identical save that they are connected to different buses. Thus this test is representative of the overheads in accessing the slow bus.

Writes to the data memory completed in 270 ms (15 ns/access), while writes to the instruction memory were completed in 526 ms (29 ns/access). This is consistent with our expectations since the ISTORE interface uses the slow buses while the DMEM interface is connected to the fast buses as shown in Fig. 5.1. The performance disparity between DMEM and ISTORE is a good indicator that bus splitting improves the performance of commonly executed instructions at the expense of rarely used instructions.

The time between event arrival and ULSNAP reacting, i.e. waking up from idle, is 6.5 ns. Upon receiving an external message or a timer event, full control is transferred to the core from the Event Handler within 14.8 ns. The first instruction starts execution within 40 ns. Note that these latencies are from SPICE simulation of extracted layout with full parasitics, since they are not directly observable on our test setup. In comparison, some processors in Table 5.2 incur in lengthy wake-up penalties. For example, the processor in [11] needs to follow a wake-up sequence consisting of 4 steps: i) turn on linear voltage regulators, ii) speed up the switched capacitor clock network, iii)enable memories and iv) transfer control to processor. Even though no reports wake-up time. This wake-up sequence takes on the order of 130 to 150 clock cycles, this latency was added to the execution time while evaluating the lifetime models reported in Fig. 5.12.

Fig. 5.13 shows the power envelope of an encryption benchmark (TEA) that is representative of the benefits of ULSNAP's event-driven design. This benchmark receives (Rx) 4 kB data from the serial interface, encrypts the data, and transmits

(Tx) the result over the serial interface. During the Tx and Rx phases the power consumption is only 22 µW. When all the data is available, encryption runs at full throughput (93 MHz). After transmission it naturally goes into a deep sleep mode and uses only 9 µW. The average power consumption of ULSNAP on this benchmark is only 98 µW.

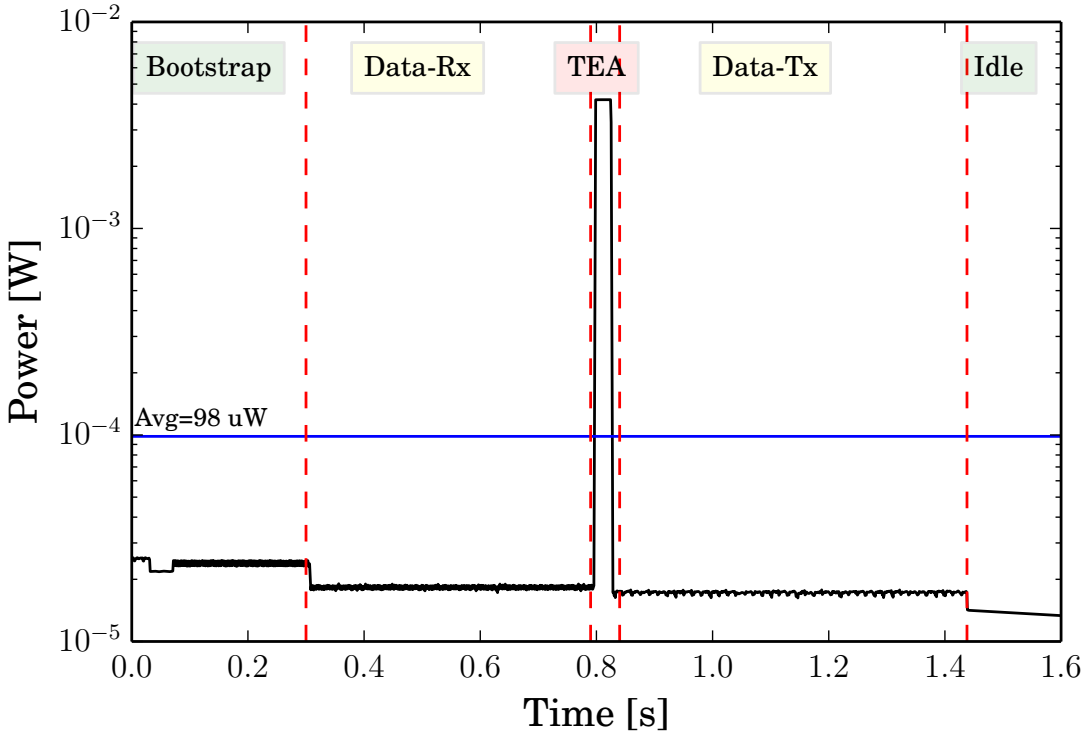


Figure 5.13: Power profile of an encryption benchmark (TEA) [47]

Note that the TEA benchmark is not annotated with power or sleep directives. In fact, the programmer need not explicitly define a sleep mode at all. The trace in Fig. 5.13 illustrates that ULSNAP will automatically scale power usage with activity.

5.5 Improving Sensor Node Lifetime

ULSNAP is a good fit for sensor network motes with bursty, computationally intensive workloads. Figure 5.13 shows how it dynamically scales its throughput to maintain the lowest possible power envelope at all times without programmer effort. Furthermore, in Fig. 5.12 we see that for computationally intensive workloads, ULSNAP is the only processor that is able to process a fast periodic stream of events in a timely manner. On the other hand, Fig. 5.12 shows a big disparity between the node lifetime between ULSNAP and other microcontrollers when the event inter-arrival time is greater than 40 s.

As the event inter-arrival time increases, the sensor mote spends most of the time in a quiescent state. Consider a 300 s(5 min) inter-arrival time, at that point, a microcontroller spends 0.033 % in active mode and 99.967 % of the time in idle mode.

Since WSN workloads are bursty, we need to focus on reducing the static power on the idle power state to improve overall mote lifetime. The difference between measured and simulated static power is shown in Fig.5.14. We attribute the big gap between the two results to the transistor model limitations and a large variance in the measured results.

Fig. 5.15 shows the power consumption of our test chip for the ULSNAP microcontroller. The static power is dominated by the memory (98 %). While it is not surprising that memories dominate the static power consumption, the absolute value of static the power consumption in the memories consume was unanticipated. In the memories, the bitcell arrays account for 99 % of the static power consumption. Since most static power dissipation is in the memories, the use the power

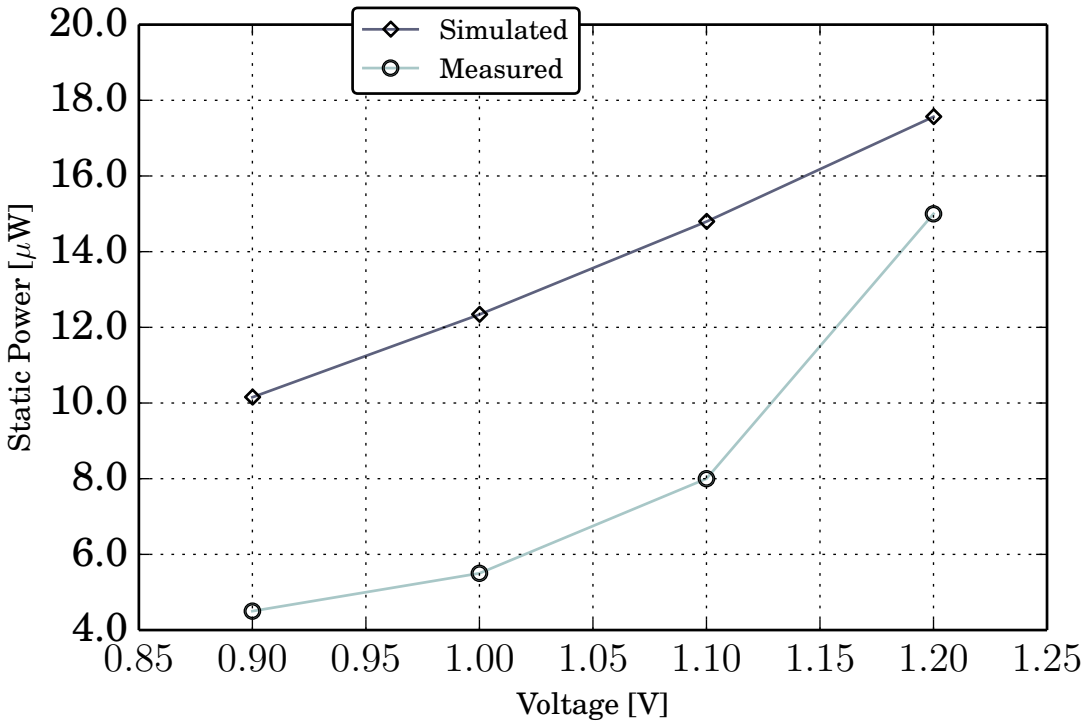


Figure 5.14: ULSNAP simulated and measured static power consumption

gating techniques discussed in Sec. 4 would not improve the lifetime (could only reduce 0.8 % of static power consumption). In order to improve mote lifetime, we should first consider reducing the static power contribution of the main component: the memory banks.

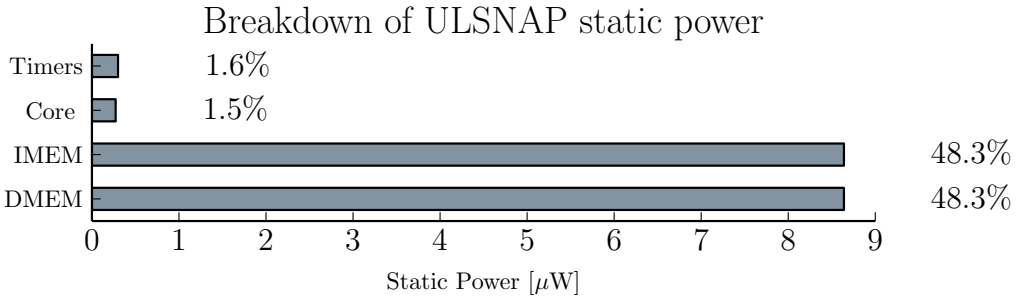


Figure 5.15: Breakdown of ULSNAP's static power at nominal Vdd (1.2 V)

The static power consumption of the core datapath is a mix of the diverse elements that comprise it. The breakdown of the static power consumption of the

core datapath is shown in Fig. 5.16. The fast execution units (`BRANCH`, `LOGIC`, `ARITH`, `DMEM`, `SHIFT`) account for 59 % of the static power consumption of the core.

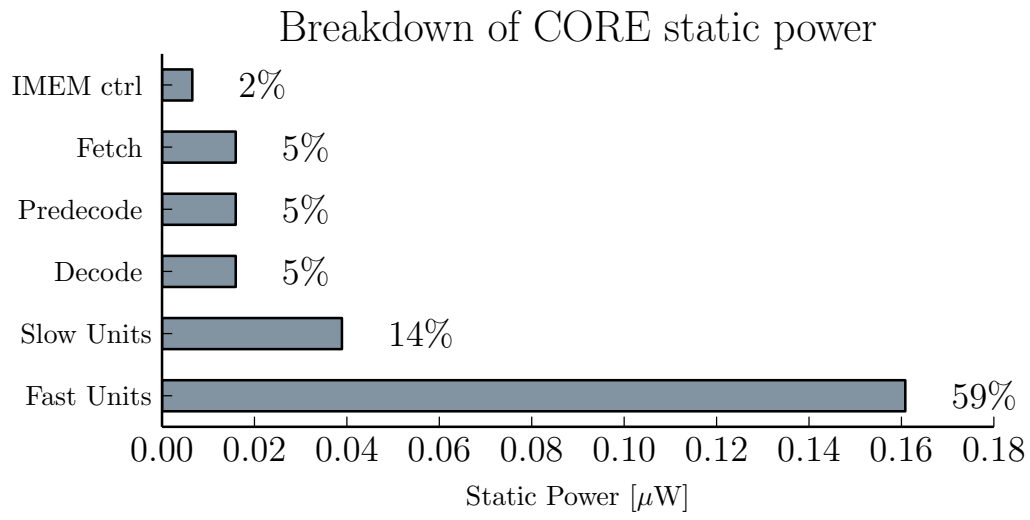


Figure 5.16: Breakdown of core's datapath static power consumption

Figure 5.17 shows the static power consumption of an 8 bank, 4 kB SRAM memory with different bitcell configurations. In the baseline configuration of Figure 5.17, the length of all transistors is $L_{design} = L_{min} = 140nm$, The access transistors have a $W = 140nm$ and the inverters have a $W_p/W_n = 180nm/200nm$. The sizing of our baseline configuration these transistors is similar to the ones used in [18,62]. The bitcell fabricated in our testchip, relies on long channel devices to reduce leakage: $W = 120nm,\ L = 0.140nm$ for the access transistors and $W_p = W_n = 120nm$, $L_p = 310nm,\ L_n = 200nm$, for the inverters. Fig. 5.17 clearly shows a reduction between our configuration and the baseline, but the use of HVT transistors provides an exponential decrease in static power compared to the memory banks that used standard-V_t transistors. The bitcell configuration that has less static power consumption is the one that uses long-channel configuration and HVT transistors: 460 nW at 0.9 V.

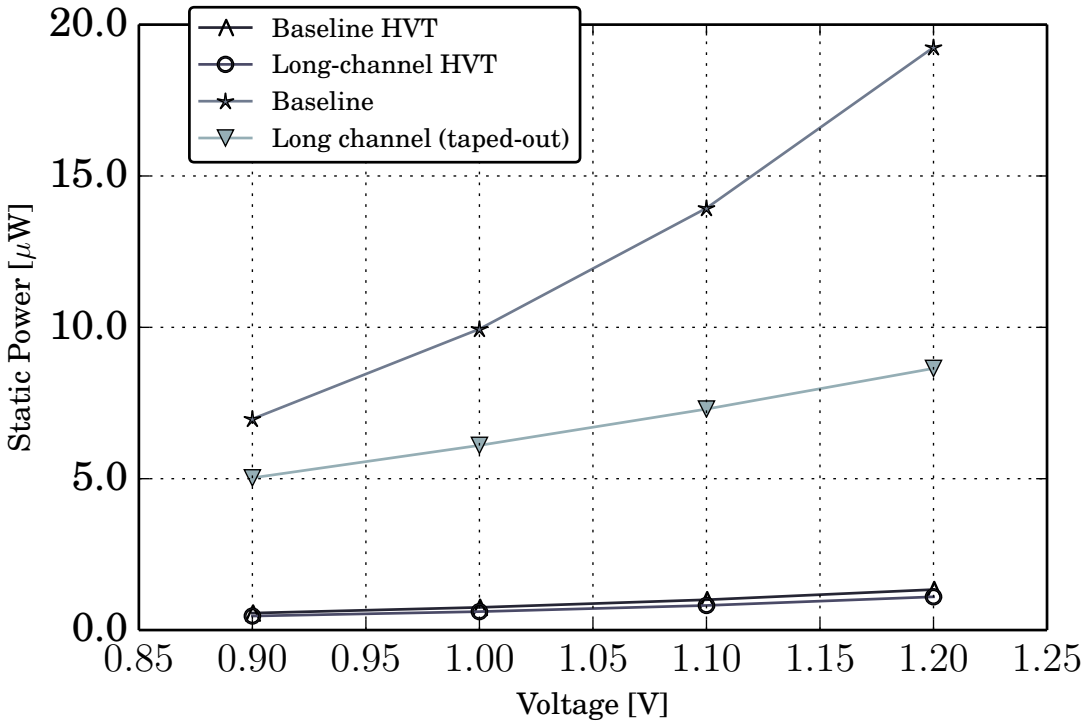


Figure 5.17: Static power consumption of a 4 kB memory bank using multiple bitcells configurations

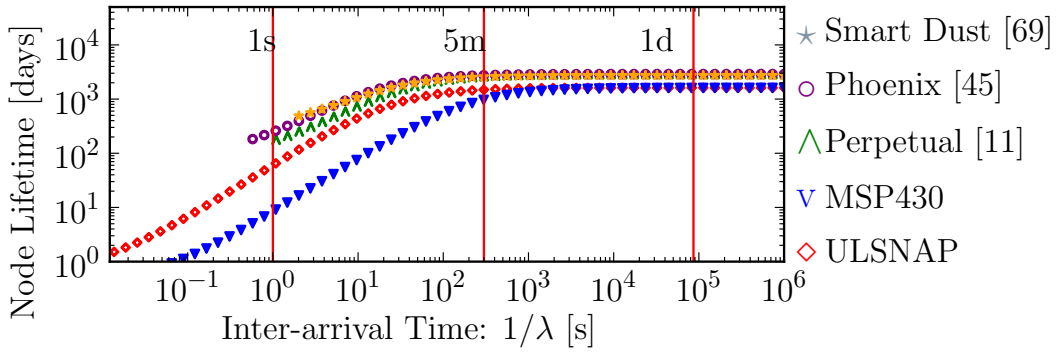


Figure 5.18: Lifetime comparison between motes with ULSNAP and other microcontrollers as function of inter-arrival time. We assume that we are using a 35 mAh, 3 V CR1220 coin cell battery

5.5.1 Near Threshold voltage scaling

There is a vast body of research that shows ultra-low energy operation is achieve by reducing the supply voltage to a particular point. As the supply voltage is reduced, the dynamic component energy reduces monotonically. The optimum energy point is achieved whenever the leakage component of energy dominates, as it has a global minimum above the functional minimum voltage [43].

On one hand, the correctness of a QDI circuit does not rely on delays of wires or gates. Hence, QDI circuits *should* automatically adjust to variations on gates and wires delay. On the other hand, the lowest digital supply voltage of our chips is 0.8 V[2]-0.9 V[3], however at 0.9 V we found that all our chips are functional.

The ideal voltage for minimum energy consumption on CMOS circuits one that is is above the nMOS transistor threshold voltage. This mode of operation is commonly called Near Threshold Voltage (NTV). Unfortunately, when reducing the supply voltage, the devices have a higher susceptibility to parametric variation—i.e. the device parameters' deviation from their nominal specifications.

By using extensive extracted SPICE simulations[4], we concluded the following factors contributed to the failure of the cells at supply voltages lower than 0.85 V:

- *Staticizers.* The production rules for an operator with a pullup network *pun*, pulldown network *pdn*, and output node z are given by $pun \rightarrow z\uparrow$ and $pdn \rightarrow z\downarrow$. Such an operator is non-interfering and *combinational* if $pun \equiv \neg pdn$. The weaker constraint of $pun|\ pdn \equiv \mathbf{true}$, denotes a non-interfering *dynamic* operator. Adding a staticizer (or keeper) to the output

[2] 70%chips work
[3] 100% working chips
[4] Lumped intrinsic and coupling model

node, z, of a dynamic operator ensures the output is always driven. A detailed analysis of these operators is can be found Sec. 4.1.

The added circuitry is a feedback inverter with input $z\_$ and output z. The problem with this solution is that in order to flip the stat of node z, the guards of the production rule need to fight the weak inverter. For example, given the initial state of $z\downarrow$. If $pup \equiv \mathbf{True}$ then the rule $pup \rightarrow z\uparrow$ needs to fight the weak rule $z\_ \rightarrow z\downarrow$. This means that the sizes of the staticizer relies on two conflicting sets of optimization constraints on the weak feedback inverter.

In the NTV regime, the requirement to satisfy the two-sided inequality becomes very difficult. In the case of the ULSNAP microcontroller, these errors manifested as stuck-at faults and incomplete transitions. The problem was exacerbated by the use of multi-threshold CMOS devices.

- *Transistor sizing and operator topology.* ULSNAP sizing was done using a semi-automatic way. This involved multiple iterations that achieve an energy local minimum point at nominal Vdd. Due to design time constraints, we were not able to perform a thorough verification, in particular at lower voltages.

  An example of an incorrectly sized PR is shown in Program 5.5. The pull-down network of $\_z$ is undersized, and the combinational feedback is oversized. This operator failed due to a combination of *self-overloading* and the lack of drive-strength of the pull-down network.

- *Number of Transistors in series.* We found a lone production rule with 6 transistors in series in the pull-down network. By empirical testing, we found a reasonable number of transistors

Program 5.5 Incorrectly sized production rule. The values between $\langle\cdot\rangle$ represent the sizes of transistors as multiples of lambda

$pcgh\langle 9\rangle \wedge \mathrm{F} \rightarrow \_z\downarrow$
$\neg pchg\langle 10\rangle \rightarrow \_z\uparrow$

$\_z\langle 10\rangle \rightarrow z\downarrow$
$\neg\_z\langle 10\rangle \rightarrow z\uparrow$

#Combinational Feedback
$z\langle 4\rangle \wedge pchg \rightarrow \_z\downarrow$
$\neg\mathrm{F}\langle 10\rangle \wedge z \rightarrow \_z\uparrow$

Something important to note, is that all circuit failures were due to operator failures. These operators break the assumptions of logic and monotonicity that QDI circuits rely on.

NTV would also improve ULSNAP's system lifetime. To that effect, we fixed all failing gates all gates that fell in aforementioned categories. Afterwards, we performed a SPICE simulation to get the static power consumption and estimate the lifetime of ULSNAP when the supply voltage is reduced to 0.7 V during the quiescent state. We chose 0.7 V since it is greater than SRAM retention voltage and also greater than the failure voltage of the gates in the datapath, hence computation can resume without exercising Reset. To evaluate the lifetime at 0.7 V, we kept the rest of the parameters the same as the ones discussed in . Fig. 5.19 shows the lifetime when ULSNAP is supply is reduced at 0.7 V during the quiescent state. ULSNAP asymptotically approaches a lifetime of 6 years, an almost two-fold improvement from the MSP430 microcontroller, The microcontroller that has best lifetime (7.7 year) is the the Phoenix microcontroller [45].

More recent research developed iterative algorithms to search for the most energy efficient operating point given a set of constraints on the digital noise mar-

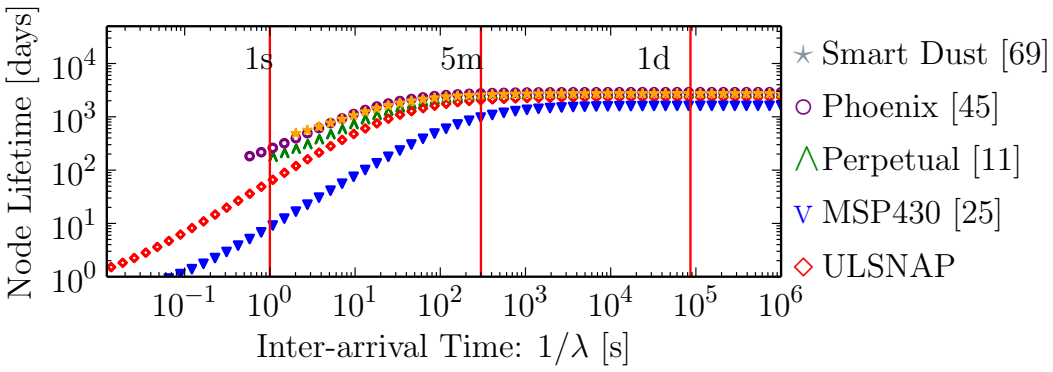
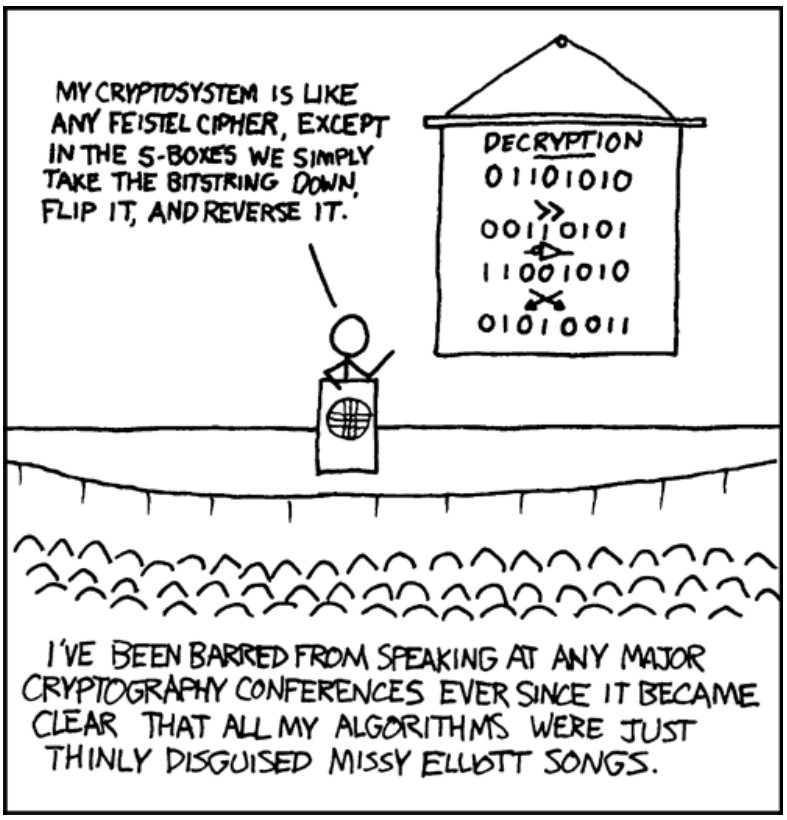


Figure 5.19: Lifetime comparison between motes between ULSNAP and other microcontrollers as a function of inter-arrival time. $\bar{T_{ULSNAP}} = 10ms$

gins [29]. We intend to develop a similar tool set to be used in the sizer that we used for the ULSNAP microcontroller.

CHAPTER 6

ENCRYPTION IN WIRELESS SENSOR NETWORK NODES



Wireless Sensor Networks (WSNs) are becoming more prevalent in a myriad of applications ranging from medical monitoring devices to industrial control systems. ULSNAP's advanced circuit and architectural techniques provide increased processing horsepower for a minimal increase in energy consumption, enabling more complex operations on collected data to be performed locally. This reduces the duty cycle of energy-hungry communication systems on motes. However, any data or computed results must be eventually transmitted wirelessly for remote collation and additional processing. This transmitted information is oftentimes sensitive and should be kept confidential. While we can and should enforce strict security policies at either end of the wireless link, encrypting the data being sent is arguably the first line of defense to protect information confidentiality [53].

This poses two questions to WSN designers: 1) Which encryption protocol to implement? and 2) How best to implement the chosen protocol given the design constraints? Implementation is further complicated by the unique challenges of the WSN encryption design space, in particular the need for small energy envelope. Typical WSN mote activity patterns are "bursty," i.e. long quiescent periods followed by a brief, highly-active period. As a result, minimizing both active *and* idle power is quite important, muddying the traditional trade-offs between application specific logic and the general-purpose processing available on a WSN.

In this section, we evaluete our WSN by contrasting two platforms: the MSP430 (CC430F6137) [25] and ULSNAP [54], representing the state-of-the-art in industry and academia, respectively.

6.1 Background

A cryptographic system is one of the main supporting tools that enables the confidentiality of sensitive data traveling across an unsafe channel. A generic cryptographic system can be modeled as a 5-tuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$, where $\mathcal{P}$ and $\mathcal{C}$ are the plaintext and ciphertext, $\mathcal{K}$ is the cipher key, and $\mathcal{E}$ and $\mathcal{D}$ are the encrypting and decrypting functions [7].

6.1.1 Cipher Functions

Most modern cryptographic systems implement their encryption ($\mathcal{E}$) and decryption ($\mathcal{D}$) functions with relatively complex algorithms. Generally, these algorithms can be separated in two different categories:

1. *Stream Ciphers* use a pseudo-random keystream based on the key ($\mathcal{K}$) to operate on each individual character of the plaintext or ciphertext.
2. *Block Ciphers* operate on fixed length chunks of the plaintext or ciphertext called *blocks*. Typical operations on the blocks are transposition or substitution.

Stream ciphers are considered safer than their block cipher counterparts only if the length of the message is less than the key length, i.e. $|\mathcal{P}| \leq |\mathcal{K}|$. Thus, stream ciphers are a rather impractical choice for many designs, particularly in environments with limited resources like a WSN. For this reason, we will concern ourselves primarily with block ciphers.

6.1.2 Modes of Operation

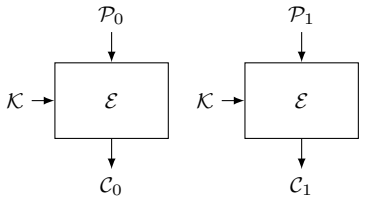
Block ciphers, by construction, operate on fixed length sections of the plaintext or ciphertext. In the case where the message is smaller than the block size, we can encode/decode by appropriately padding the message. However, in the case where a message exceeds the block size, we have a choice between several different *modes of operation* [51]. In all of the modes, there is a block cipher encryption step, which uses the key ($\mathcal{K}$) to encrypt a block's worth of data. This block encryption step is defined by the cipher and is the same across all modes of operation. What differentiates the modes of operation from one another is the inputs and outputs to the block encryption step:

- *Electronic Codebook (ECB)* — The input to the block encryption step is a plaintext block and the output the corresponding ciphertext block, as shown

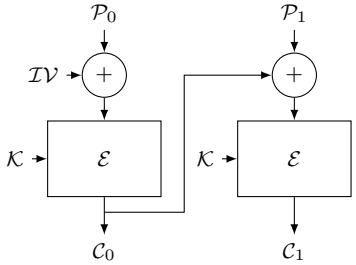
in Fig. 6.1a. Each block's ciphertext can be independently computed from each block's plaintext.

- *Cipher Block Chaining Mode (CBC)* — The input to each block encryption step is the XOR of a plaintext block and the previous ciphertext block, as shown in Fig. 6.1b. CBC combines ideas from both block and stream ciphers, effectively mimicking a stream cipher that operates at the block level as opposed to the character level. The first plaintext block is typically XORed with an Input Vector ($\mathcal{IV}$). Care should be taken when selecting an $\mathcal{IV}$, as this presents a potential attack vector on the cryptographic system.
- *Cipher Feedback Mode (CFB)* — The output of each block encryption step is XORed with the corresponding plaintext block, as shown in Fig. 6.1c. The post-XOR data is the ciphertext, which is passed to the next block as the input to its block encryption step. This effectively emulates a stream cipher using block cipher primitives. As there is no ciphertext for the first block, we must make use of an $\mathcal{IV}$, just as in CBC.
- *Output Feedback Mode (OFB)* — OFB is very similar to CFB in that it effectively is also a stream cipher. The key difference, shown in Fig. 6.1d, is that the input to the next block's encryption step is the output of the previous block's encryption step instead of the previous ciphertext.
- *Counter Mode (CTR)* — CTR can be viewed as a compromise between the various feedback or chaining modes and ECB. Like CFB and OFB, the XOR of the plaintext and the output of the encryption step is the ciphertext. However, as seen in Fig. 6.1e, the input to each block encryption step is a different $\mathcal{IV}$, typically implemented as an incrementing counter. While CTR is still a stream cipher mode similarly to CFB and OFB, the use of a counter breaks the dependency on the previous block encryption step.

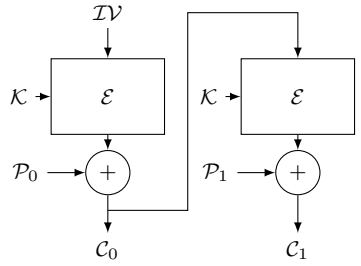
- Authentication Modes — Authentication modes such as *Galois/Counter (GCM)* and *Counter with CBC-MAC (CCM)* offer an authentication tag in addition to ciphertext to provide confidentiality, integrity, and authenticity assurances on the encrypted data. Of course, these additional features come with additional overheads. Both the GCM and CCM modes are based on the CTR mode of operation.



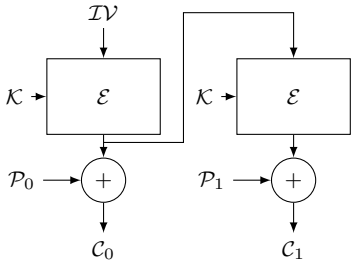
(a) Electronic Codebook (ECB)



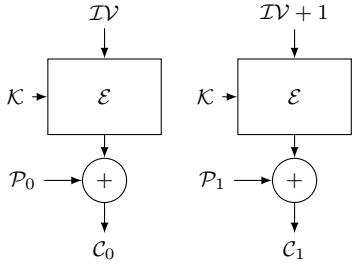
(b) Cipher-Block Chaining (CBC)



(c) Cipher Feedback (CFB)



(d) Output Feedback (OFB)



(e) Counter Mode (CTR)

Figure 6.1: Common Cipher Modes of Operation

6.1.3 Cipher algorithms

There are a number of possible choices for cipher algorithms. To aid in our design space exploration, we leverage the analysis done by the US government on several of the leading candidates. As of December 2013, the US Federal government requires a minimum security strength of 112 bits[1] for all sensitive transactions [52]. As of 2014, the US National Institute of Standards and Technology (NIST) approves 3 block ciphers for federal use with certain restrictions:

- *Advanced Encryption Standard (AES)* [50] is approved for both non-classified and classified information by the US Government. Typical key lengths are 128, 192, or 256 bits. As the only publicly known and computationally feasible attacks on AES are side-channel attacks [8,56,64], NIST has rated AES to provide at least 128 bits of protection [52].
- *Triple Data Encryption Algorithm (TDEA)* [48], also referred to as Triple Data Encryption Standard (Triple DES), has a 168-bit key length. However, due to several weaknesses, NIST has estimated TDEA provides only 80 bits of security[2] [52]. TDEA is also relatively complex in comparison to other block ciphers.
- *Skipjack* [49] was specifically designed for efficient computation and low memory footprint [32]. Attacks of the Impossible Differential Cryptanalysis (IDC) type are only known on 31 of the 32 rounds and they are only marginally faster than the brute force attack [6].

[1] With the exception of digital signatures, which are allowed to use 80 bits of security strength

[2] Provided that the attacker has approximately 2^{40} ($\mathcal{P}$, $\mathcal{C}$) pairs encrypted with the same $\mathcal{K}$ vector.

6.2 Sensor Network Encryption

The WSN design space presents a number of additional design challenges for cryptographic systems, in particular maximizing energy efficiency. One cannot simply examine a sensor mote in isolation, however. The choice of cipher algorithm as well as cipher mode of operation is largely dependent on system-level considerations. Additional design choices include the Open Systems Interconnection (OSI) model layer at which encryption is implemented, software versus hardware versus hybrid software/hardware implementations, as well as cipher key length.

6.2.1 Cipher Selection

Tables 6.1 and 6.2 list different encryption implementations in existing WSN solutions, the ciphers used, as well as the mode of operation for the link/network layer and session layer, respectively.

Table 6.1 and Table 6.2 show that AES, TDEA, and Skipjack are quite popular for both network/link and session layer implementations. RCx cipher algorithms, while secure and popular, do not offer significant competitive advantages with respect to AES.

Skipjack is attractive, given its ease of implementation and small memory footprint, and may be appropriate for applications where security needs are less stringent. However, NIST Special Publication 800-131A has recommended that Skipjack be phased out in 2010 except for legacy applications and some variants of TDEA be phased out by 2015. AES is the only symmetric cipher deemed as acceptable without restrictions by NIST 800-131A [52]. Given all the constraints,

AES is our cipher of choice. This choice is consistent with existing comprehensive security frameworks for WSNs such as TinySec [28] and MiniSec [35].

Table 6.1: Link and Network Layer

Protocol	Cipher	Mode of Operation
ZigBee	AES	CTR, CBC, CCM
Z-wave	AES	CFB, OFB
TI (MSP430,CCX)	AES	Various
Clipper based	Skipjack	-
TinySec [28]	Skipjack	CBC
MiniSec [35]	Skipjack, AES	CBC
SNEP (SPINS) [57]	RC5	CTR

Table 6.2: Session Layer

Library	Cipher	Mode of Operation
GPG	AES, TDEA, CAST5, 2F, Camelia	CFB
IPSec	AES, TDEA,	CBC
SSH	AES	CBC
SSL3.0	AES, TDEA, IDEA, Cam, D, RC[2-5]	CBC, GCM, CCM
TLS2.0	AES, TDEA, IDEA, Cam, D, RC[2-5]	CBC, GCM, CCM

We can see that the choice of OSI layer to implement encryption does not strongly affect the cipher mode of operation. Of particular interest is the absence of ECB in both tables. ECB is amenable to many optimizations such as pipelining and loop unrolling. However, ECB mode leaves the system vulnerable to attacks such as known-plaintext and known-ciphertext [51].

On the other hand, CBC makes several appearances in both types of implementations. The CBC mode has a loop dependency between two adjacent blocks in the data stream with the encryption step on the critical path. Thus, when using CBC, designers must pay particular attention to reducing the latency of the block encryption/decryption step. Typical hardware implementations minimize latency

by using lookup tables (LUT) or logic arrays (PLA).

There has been some discussion regarding CTR and its derivatives, in particular regarding a perceived vulnerability due to the use of an incrementing counter. Limpaa *et al.* argue that any such vulnerability is due to a differential weakness in the block cipher used and not a valid criticism of CTR [34]. Of course, as with any stream cipher, great care must be taken to not reuse the chosen $\mathcal{IV}$ for CFB, OFB, and CTR. Ensuring this constraint is maintained across all motes in a sensor network is impractical, so in this work we primarily focus on CBC as our mode of operation, similar to TinySec [28].

6.2.2 Software Implementations

Software implementations of cryptographic systems can be quite troublesome. Programmers face the traditional problems of bugs, maintenance, and optimizations, but the negative effect of these issues is higher than usual. Optimizations can lead to potential attack vectors, bugs can lead to outright breaches of confidentiality, and maintenance patches must be carefully vetted to avoid introducing any additional vulnerabilities.

The benefit to a software implementation is mutability—changes can be made to applications deployed in the field to correct bugs and close off potential avenues of attack. Mutability is a clear advantage over a hardware implementation, but it comes at a loss in active energy use and throughput. To combat this, especially in the energy-starved WSN application space, designers must make careful optimizations to reduce energy consumption of the cryptographic system, which can often increase complexity of the software. As such, there are conflicting goals in a

software implementation: throughput/energy efficiency and software simplicity.

6.2.3 Hardware Implementations

In general, hardware cryptographic systems offer energy and throughput advantages at the cost of being immutable. One of the primary enablers for high-throughput hardware implementations is pipelining. Non-pipelined datapaths execute the entire computation as an atomic operation, at least from the point of view of any control structures. In contrast, a pipelined datapath breaks the computation into sequential operations that can each operate independently on data. Traditionally, pipelining improves throughput at the cost of latency through the datapath. Feedback is also difficult to incorporate into a pipelined datapath, which is a significant implementation hurdle for many of the cipher modes of operation detailed in Sec. 6.1.2.

One of our evaluation microprocessors, ULSNAP, makes use of asynchronous circuits [54], which we briefly cover here. While there are several families of asynchronous circuits, ULSNAP makes use of Quasi-Delay Insensitive (QDI) circuits, which are the most robust to variations in process, voltage, temperature, and timing. QDI circuits are built with Martin Synthesis [39], which is a procedure that breaks apart a computation into fine-grained hardware processes that communicate over point-to-point delay insensitive channels. Instead of using a global clock to synchronize actions and flip-flops as storage elements, QDI circuits use channels for local, between-process synchronization and represent data as tokens traversing these channels.

The logic that makes up each QDI hardware process and the QDI channels

themselves are robust to arbitrary gate delays. As a result, QDI circuits are intrinsically tolerant to process, temperature, and voltage variations. QDI circuits are also naturally event-driven, waiting in a quiescent state with no switching activity until a data token arrives. This is the equivalent of perfect clock-gating in a synchronous system—inactive processes consume only leakage current. Since encryption is typically only active during data transmission in the WSN application space, an asynchronous circuit implementation benefits from effectively instantaneous wake-up/sleep time and a perfect clock-gating implementation. The ULSNAP processor used in our evaluation has this property [54].

6.3 AES Implementation

AES has become the industry standard in applications ranging from SSL to storage media encryption. For this reason and those outlined in Sec. 6.1.3, we keep the focal point in AES within the context of WSNs. Here, we provide a brief overview of AES, focusing on the logical breakdown of the cipher in preparation for discussing our hybrid implementations in Sec. 6.4.

AES uses 128-bit blocks and either 128-, 192-, or 256-bit keys ($\mathcal{K}$). Typically, the blocks are further organized into a 4×4 matrix of 8-byte elements. This matrix is oftentimes referred to as the *state*. Each block of plaintext ($\mathcal{P}_i$) and ciphertext ($\mathcal{C}_i$) is 128 bits to match the AES block size.

Internally, an AES block cipher encryption/decryption step is implemented as a network of substitution and permutation operations wrapped in a loop or *round*. The number of round iterations is dependent on the key length—10, 12, or 14 rounds for 128-, 192-, and 256-bit keys, respectively. This structure is then

organized into a larger structure to implement the desired cipher mode of operation. The initial key is expanded into multiple 128-bit round keys. This step is called the *key schedule*, and can be pre-computed or computed on-the-fly for each encryption. Typically, each iteration of a round performs 4 basic operations on the elements of the 4×4 state matrix:

6.3.1 Add Key (AK)

Perform a bitwise XOR of each byte in the state matrix and the corresponding byte of the current input block.

6.3.2 Byte Substitution(BS)

Substitute each byte of the input block with one from a lookup table using the non-reversible, non-linear mapping provided by a Galois Field, typically $GF(2^8)$.

6.3.3 Shift Rows(SR)

Cyclically left shift each row of the state matrix. The shift amount for the n^{th} row is n bytes, assuming we index from zero.

6.3.4 Mix Columns (MC)

Apply an invertible linear transform to each state matrix column. The input and output of the transform function are both 32-bits wide.

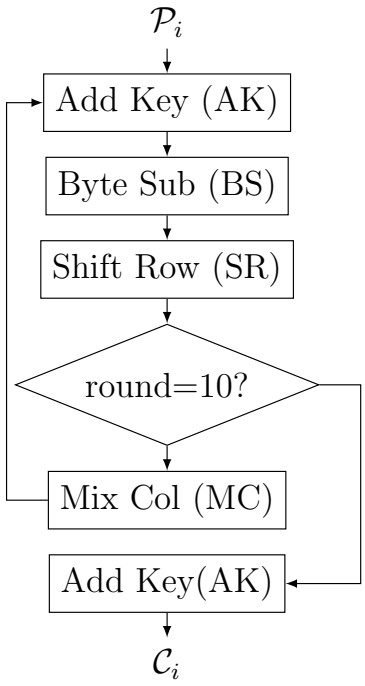


Figure 6.2: AES Block Cipher

6.3.5 Block Cipher

Fig. 6.2 shows the dataflow diagram for a complete execution of a 128-bit key AES encryption. The contents of Fig. 6.2 are then wrapped with the appropriate control structures to implement the desired cipher mode of operation. Effectively, Fig. 6.2 is an expansion of the $\mathcal{E}$ blocks in Fig. 6.1.

As of now, many if not all WSN motes are designed with a single-threaded execution model to reduce energy consumption. Thus, the cost of ciphering a single block governs the overall encryption performance for all modes of operation. The traditional optimizations are still available to the programmer, e.g. loop unrolling, macro insertion, and inline assembly code. However, the embedded programming environment presents some unique challenges, in particular the limited memory

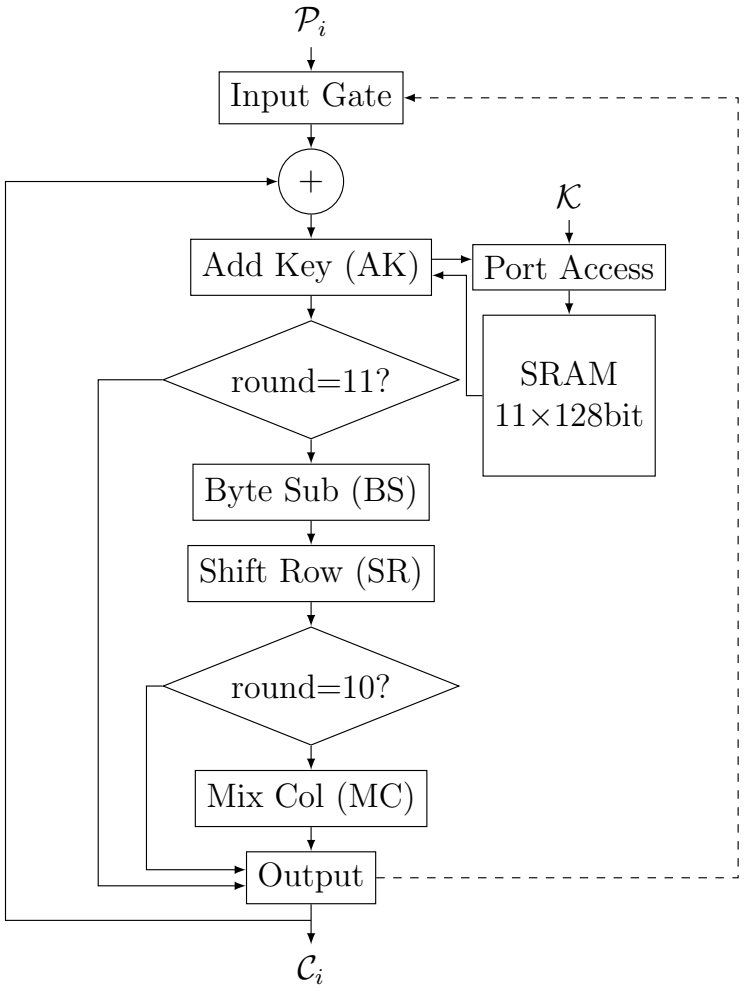


Figure 6.3: Non-pipelined AES implementation

space, even for programs. We compare complete software implementations of AES against hardware and hybrid implementations in Sec. 6.4.

If we are willing to invest in a hardware implementation, we have the opportunity to depart from the single-threaded execution model enforced on software implementations by the low-energy WSN environment. Completely unrolling the loop computation is possible on a system such as an FPGA [67], but this is too expensive in both energy and area for the WSN space.

Another available hardware optimization is pipelining the computation, but

this is only useful for modes of operation without data dependencies, i.e. ECB and CTR. As discussed in Sec. 6.2.1, we choose CBC over ECB and CTR due to their insecurity and unsuitability for the WSN space, respectively. Due to the feedback or loop dependency in modes like CBC, deep hardware pipelining represents a significant energy and area overhead for little to no performance benefit. In practice, some degree of pipelining is still desired to logically separate the various stages of encryption, reduce signal fanout, and ease system design.

QDI circuits, as described in Sec. 6.2.3, have the additional and automatic benefit of decoupling pipeline occupancy from pipeline depth. In other words, an n stage asynchronous pipeline can potentially support up to n data tokens in flight but will gracefully handle as few as 0 or 1 tokens without necessitating the injection of NOPs. Since QDI circuits effectively offer fine-grained clock-gating, this alleviates the energy overhead of pipelining.

Fig. 6.3 illustrates our encryption engine hardware implementation. While it is implemented at the circuit level as a pipelined system—each box represents a pipeline stage, we treat the system as an un-pipelined functional unit. The *Input Gate* process only allows a single token to enter the pipeline, waiting for a "done" signal from the *Output* process before allowing another token in.

We implemented AES using the CBC mode of operation, so the next step is an XOR with the output or $\mathcal{IV}$ as appropriate. The unrolled encryption key is stored in an SRAM memory that can be written externally by the user. The SRAM is readable only by the *AK* process. All access to the SRAM is controlled via the *Port Access* process, which arbitrates between key writes and accesses.

It is worth noting that while the encryption of one message using AES CBC is

not pipelined, the encryption of *multiple* messages can be pipelined. By modifying the *Input Gate* and *Output* processes to properly interleave the plaintext and ciphertext blocks of the various messages, we can have multiple in-flight encryptions. The maximum amount of in-flight messages is limited by the maximum occupancy set by the circuit-level implementation of the system shown in Fig. 6.3.

6.4 AES Evaluation

Two key metrics are of paramount importance while building a sensor network node: mote lifetime and performance. Increasing application complexity in WSNs has forced increased throughput requirements on all steps in the typical WSN computation: gathering data, processing data, encrypting results, and sending results. Thus, maximizing the throughput of the encryption step without adversely affecting mote battery life is of great interest. In the context of WSN cryptographic system implementations, particularly in hardware, memory usage and area are also metrics of interest.

To evaluate impact of the WSN lifetime with the addition of AES to our system, we can modify the TX state S_3 in the semi-Markov chain described in Fig. 3.2 as shown in Fig. 6.4. The TX state, S_3 is now comprised of an embedded chain of an encryption state S_E and data transmission state S_T.

Assuming that $X \gg Y, Z$, we approximate the average sensing time X as the inter-arrival time $1/\lambda$. We also expand the power state S_3 into its embedded Markov chain of S_E and S_T, which means that the mean time spent in S_3 can be expressed as the sum of the mean times spent in S_E and S_T, i.e. $Z = T_E + T_T$. We can thus rewrite Eq. (3.11) as:

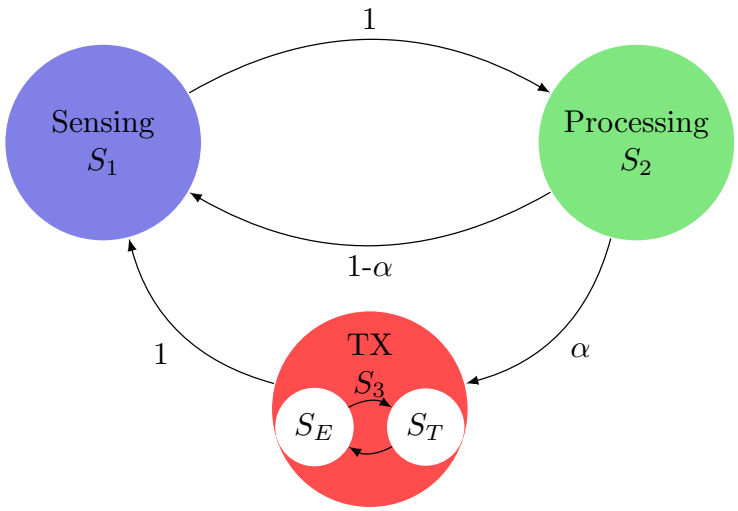


Figure 6.4: Semi-Markov Chain for cryptographic WSN Mote

$$t_{\text{life}}(\lambda) \leq \frac{E_{\text{total}}(1+\lambda(Y+\alpha(\bar{T_T}+\bar{T_E}))}{P_1+\lambda(\bar{\mu_2}P_2+\alpha(\bar{T_T}P_T+\bar{T_E}P_E))} \tag{6.1}$$

We use Eq. (6.1) in this section to compute mote battery lifetime, a key figure of merit for our evaluation.

In our study, we benchmarked and analyzed software, hardware, and hybrid AES implementations for two microcontrollers: the TI MSP430, version CC430 [25], and ULSNAP [54]. MSP430 cores are more power efficient than Atmel's ATMega counterparts and are widely used, both by professional engineers and electronics hobbyists. Furthermore, the MSP430 chip provides a completely in-silicon solution for AES. ULSNAP is a microcontroller design from academia targeted for the WSN space. ULSNAP employs a number of advanced techniques at both the circuit and microarchitectural levels to improve energy efficiency while maintaining performance [54].

The MSP430 CPU implements a 16-bit, single-pipeline Von Neumann architecture with a modern RISC ISA. ULSNAP is also 16-bit and has a MIPS-like RISC

ISA, but implements a Harvard architecture. More details about ULSNAP's architecture can be found in Chapter 5 and Appendix B. Again, while ULSNAP does not have explicit active power modes, by changing the operating voltage the user can choose high-performance and low-power characteristics. At the maximum operating voltage of 1.2 V, ULSNAP runs at 93 MHz and draws 3.45 mA—47 pJ per operation. By lowering the operating voltage to 0.95 V, we can reduce ULSNAP's performance and current consumption to 47 MHz and 1.3 mA, respectively. Energy consumption for ULSNAP at 0.95 V is 29 pJ per operation on average.

With respect to our model described in Sec. 3.2, we assume regularly spaced sensor events, each of which has probability α of causing the transmission of a single encrypted packet after being processed. The total size of each packet has a payload of 127 B. This payload is the maximum allowable data to be transmitted in a Zigbee packet as defined by the IEEE 802.15.4 standard. The transmission time of a packet, T_T, is set by the transmission rate. We assume that our motes use the TI-CC1101 transceiver which offers 500kbps of bandwidth at 55 mW [24]. Using this transceiver is natural fit for the MSP430, and we assume ULSNAP can be paired with the CC1101 or an equivalent radio. Transmitting a 42 B packet takes $T_T = 672\,\mu s$.

We assume the average processing time for each event to be $Y = 1.1$ ms, which is the average completion time for the statistical benchmark set from the SenseBench suite running on ULSNAP as described in Section 5.4.1. X and T_E are dependent on the event inter-arrival time and AES implementation, respectively. Note that encrypting the 29 B payload takes two full encryptions, which we have accounted for in our evaluation below.

6.4.1 Software-Only Approach

The best software implementations are those that are hand-tuned for a particular hardware platform. For consumer/business software, this is not always possible, but in the WSN application space the hardware platform is well-defined. In order to maximize energy efficiency, programmers can leverage different hardware features such as operating mode to reduce power consumption. The MSP430 microcontroller offers a total of 9 different power modes. Five of these modes are inactive modes where the CPU core and various peripherals can be shut off. The remaining 4 modes are active modes offering a tradeoff between active power consumption and processor performance. Table 6.3 summarizes these active modes. ULSNAP does not offer any explicit power modes, but performance scales smoothly with operating voltage due to its QDI circuit implementation. Furthermore, it naturally goes into a deep sleep mode when all events have been processed.

Table 6.3: MSP430 Active Core Power Modes

Mode	Voltage [V]	Current [mA/MHz]	Max Freq. [MHz]
0	1.8	0.160	8
1	2.0	0.190	12
2	2.2	0.213	16
3	2.4	0.225	20

In mode 3, high performance (HP) mode, the MSP430 core consumes 4.5 mA at 2.4 V, which translates to about 54 nJ per operation. When running in mode 0, the *minimum* frequency is 1 MHz. In this extreme low-energy (LE) mode, the MSP430 runs at 1.8 V and consumes 160 µA for a energy per operation of 28 pJ.

In order to compile our software implementations, we used Code Composer Studio with the MSP430 processor and the LCC compiler toolchain for ULSNAP. For the MSP430, we counted the cycles to complete a full encryption and multiplied

by the clock frequency to calculate the throughput and delay. Since ULSNAP has no clock, we measured throughput and delay for ULSNAP using a logic analyzer to measure the start and stop times for encryption.

Table 6.4: AES Software Implementations

Design	Mode	Perf. [Mbps]	Power [mW]	Energy [nJ/bit]	Memory [B]
TI-C	HP	0.102	10.8	105.4	3441
	LE	0.005	0.2	56.1	
TI-MSP430	HP	0.420	10.8	25.3	1184
	LE	0.021	0.2	13.5	
ULSNAP-C	HP	1.550	4.1	2.6	2670
	LE	0.786	1.2	1.5	
ULSNAP-O	HP	1.850	4.1	2.2	2664
	LE	0.935	1.2	1.3	

Table 6.4 shows the power and performance for four different software implementations of AES running at the high and low power modes of our test microprocessors. HP is active Mode 3 for the MSP430 and 1.2 V for ULSNAP, and LP is active Mode 0 for the MSP430 and 0.95 V for ULSNAP. TI-C and ULSNAP-C represent the same AES library written in C compiled for the MSP430 and ULSNAP, respectively. TI-MSP430 is a Texas Instruments software implementation of AES optimized for the MSP430. ULSNAP-O is an optimized software implementation of AES written by the designers of ULSNAP.

In general, ULSNAP performs better than its MSP430 counterpart. For instance, the optimized TI library provides maximum throughput of 0.42 Mbps. The ULSNAP core *quadruples* the MSP430 performance to 1.85 Mbps while using 10x *less* energy. On the other hand, the ULSNAP microcontroller uses much more memory than the TI implementations. This is mostly due to the differences in memory architecture—ULSNAP is word-aligned (16-bits) while the MSP430 allows for byte-aligned memory access.

6.4.2 Hardware-Only Approach

Table 6.5: AES Hardware Implementations

Design	Process [nm]	Perf.	Latency	Energy [pJ/bit]
[9]	130	141 Mbps	910 ns	79
[14]	350	9.9 Mbps	12.9 µs	–
[14]	350	12.8 Kbps	10.3 ms	55000
[67]	180	1.6 Gbps	80 ns	300
[46]	130	10.0 Gbps	11.3 ns	191
MSP430	–	15.0 Mbps	8.5 µs	717
ULSNAP-AES	180	907 Mbps	138 ns	34
ULSNAP-AES	90	948 Mbps	135 ns	8

We define a "hardware approach" as any implementation of AES as an isolated coprocessor. The plaintext and key are transferred to the specialized coprocessor and the ciphertext is returned. Oftentimes, the AES coprocessor will trigger an interrupt upon completing encryption, freeing the main processor to engage in another task in parallel. Many hardware implementations of AES exist, from both academia and industry. Common optimization goals are low transistor count, high throughput, and low energy. While we cannot compare all possible AES implementations, we show the reported numbers for the best-in-class implementations in Table 6.5.

A comparison of the hardware implementations in the Energy-Throughput space is shown in Fig. 6.5. The implementation in [46] offers the best throughput, delivering 10Gbps, however it uses almost 2 W of power (191pJ/b), making it a choice more suitable for high performance applications such as servers or network routers. Our design, ULSNAP-AES, delivers encryption at rates of 950Mbps, while only requiring only 8pJ/b. All designs in Table 6.5 and Fig. 6.5 compare implementations of the AES algorithm running in CBC mode.

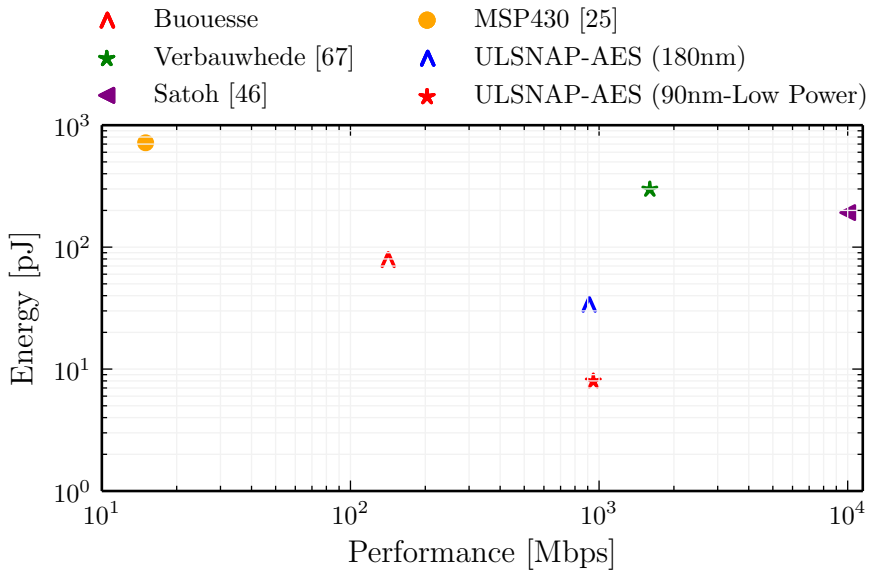


Figure 6.5: Comparison of AES Hardware implementations on the Energy-Throughput space

We augment ULSNAP with a CBC implementation of AES as described in Sec. 6.3. Our primary design goal for our implementation at both the circuit and architectural levels was minimizing energy. Our performance and throughput numbers come from transistor-level SPICE simulations, including wiring capacitance. Table 6.5 includes two different versions of our AES implementation, one in a 180 nm high-performance process and the other in an 90 nm low-power process to match ULSNAP [54]. The static power consumption of ULSNAP AES is 7.54 µW and 17.3 µW for the 180 nm and 90 nm versions respectively. The performance and static power numbers illustrate the effects of the different doping characteristics of the high-performance and low-energy processes, and the effects of scaling on leakage current.

Table 6.5 also includes the MSP430's hardware implementation of AES. Our measurements indicate that the Processing step (S_2) takes 170 clock cycles, delivering 15.05 Mbps. However, an extra 140 clock cycles are required to load the plaintext into the encryption unit, set the coprocessor interrupts, and retrieve the data from the AES coprocessor, so the *net* performance rate is 8.5 Mbps.

6.4.3 Hybrid Approach

As described in Sec. 6.3, we can logically partition the AES computation, enabling us to implement a hybrid scheme where some AES blocks are implemented in hardware and some in software. We partitioned the AES system into four parts: Loop Control (Ctrl), Add Key (AK), Byte Substitution and Shift Rows (BS), and Mix Columns (MC). We combined Byte Substitution and Shift Rows as a logical step as they are always executed sequentially, as seen in Fig. 6.3.

Table 6.8 enumerates all possible hybrid hardware/software configurations of our AES implementation, with the caveat that Ctrl is only implemented in hardware if AK, BS, and MC are also all hardware. For all software blocks we used the same AES library as used for our earlier valuation of the TI-C and ULSNAP-C software configurations. All of the ULSNAP configurations are evaluated in the 90 nm low-power process discussed earlier. Per-block measurements for both software and hardware ULSNAP implementations are available in Tables 6.6 and 6.7, respectively. As we do not have access to the details of the MSP430 hardware implementation of AES, we substitute the appropriate pieces of our AES hardware implementation for the following analysis of hybrid configurations on the MSP430.

Table 6.6: ULSNAP Software AES Blocks

Block	Mode	Delay [µs]	Energy [pJ/bit]	Memory [B]
AK	HP	3.30	106.82	452
	LE	6.54	63.05	
SUB	HP	1.25	40.55	1058
	LE	2.48	23.94	
MC	HP	3.88	125.62	552
	LE	7.68	74.15	
MEM	HP	2.14	69.24	40
	LE	4.24	40.87	

Table 6.7: ULSNAP Hardware AES Blocks

Block	Perf. [MHz]	Energy [pJ/bit]	Txr [×1000]	Delay [ns]	P_{static} [µW]
AK	370	0.140	28.65	0.21	0.47
MC	281	0.136	14.57	0.57	0.80
S-box - ROM	155	0.351	7.45	3.54	0.84
S-box - GF	192	0.750	5.49	4.83	†0.29
Ctrl			107.7		11.30

†: Encryption and Decryption

In order to incorporate a hybrid implementation into our model as described in Sec. 3.2 and throughput calculations, we define T_{hyb} as the block encryption time of our hybrid system. T_{load}, as seen in Eq. (6.2), is the inherent cost to load and retrieve data for the accelerator, which we assume is similar to the cost of accessing memory. T_{AK}, T_{BS}, and T_{MC} are the execution times for the AK, BS, and MC units, respectively. The coefficients below represent the total number of executions of each unit for a complete encryption, including the load time T_{load}:

$$T_{\text{hyb}} = 10T_{\text{load}} + 11T_{AK} + 10T_{BS} + 9T_{MC} \tag{6.2}$$

$$T_{\text{hw}} = \quad T_{\text{load}} + T_E \tag{6.3}$$

$$T_{HSH} = 10T_{\text{load}} + T_{\text{hyb}} \tag{6.4}$$

Note the difference in the coefficients of T_{load} between the hybrid execution time shown in Eq. (6.2) and the hardware execution time shown in Eq. (6.3). The T_E here accounts for the double encryption necessary to send a 29 B packet. A full hardware implementation needs to access the encryption data once whereas a hybrid implementation must access ten separate times due to the mixed hardware/software implementation. In the situation where AK and MC are implemented in hardware but BS is implemented in software, we actually incur an

additional ten memory accesses as we need to retrieve the data twice, as shown in Eq. (6.4). We refer to this combination as "hardware-software-hardware" (HSH).

Table 6.8: Hybrid AES Implementations

					ULSNAP			MSP430		
	Ctrl	AK	BS	MC	Perf. [Mbps]	$t_{\text{life}}{}^{\dagger}$ [days]	Memory [B]	Perf. [Mbps]	$t_{\text{life}}{}^{\dagger}$ [days]	Txrs [×1000]
0	H	H	H	H	57.02	165	40	20.05	161	180.70
1	S	H	H	H	6.05	288	40	2.05	275	73.02
2	S	H	H	S	2.27	304	592	0.49	401	58.5
3	S	H	S	H	2.35	373	1098	-	-	43.2
4	S	H	S	S	1.87	401	1650	-	-	28.6
5	S	S	H	H	2.24	326	492	-	-	44.4
6	S	S	H	S	1.39	347	1044	-	-	29.8
7	S	S	S	H	1.85	440	1550	0.11	567	14.6
8	S	S	S	S	1.54	478	2102	0.10	614	0.0

†: We assume that $1/\lambda = 1\,\text{min}$, $Y = 1.1\,\text{ms}$, $T_T = 672\,\mu\text{s}$, $\alpha = 0.1$, and that we are using a 35 mA h, 3 V CR1220 battery.

Table 6.8 lists our estimated performance and lifetime numbers for various hybrid implementations of AES on ULSNAP and the MSP430. Again, because we did not have access to the details of the MSP430's hardware implementation, we used the various blocks of our AES implementation for ULSNAP instead. The throughput was calculated by adding the delay of blocks from Tables 6.6 and 6.7 as appropriate. As a validation of our estimates of software block implementations, the software only approach in Table 6.8, configuration 8, matches the measured result in Table 6.4 (ULSNAP-C) to within 1 %. Similarly, the software only approach of the MSP430 matches the (TI-C) result from Table 6.4. The TI-C software implementation fuses the AK and BS steps, so we did not evaluate hybrid combinations of AK and BS on the MSP430.

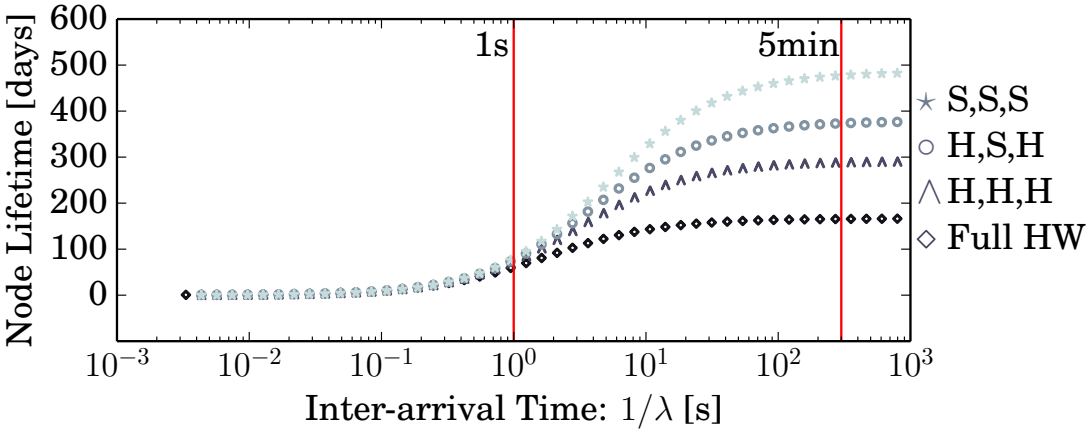
We present t_{life} in Table 6.8 as an estimate of mote battery life, following our model. We obtain the value via Eq. 3.11 obtaining T_E as discussed above and assuming an inter-arrival rate of $1/\lambda = 1\,\text{min}$. The static power consumed in state

S_1 corresponds to the static power consumption of the microcontroller plus the static power consumed by any hardware-implemented blocks as shown in Table 6.7.

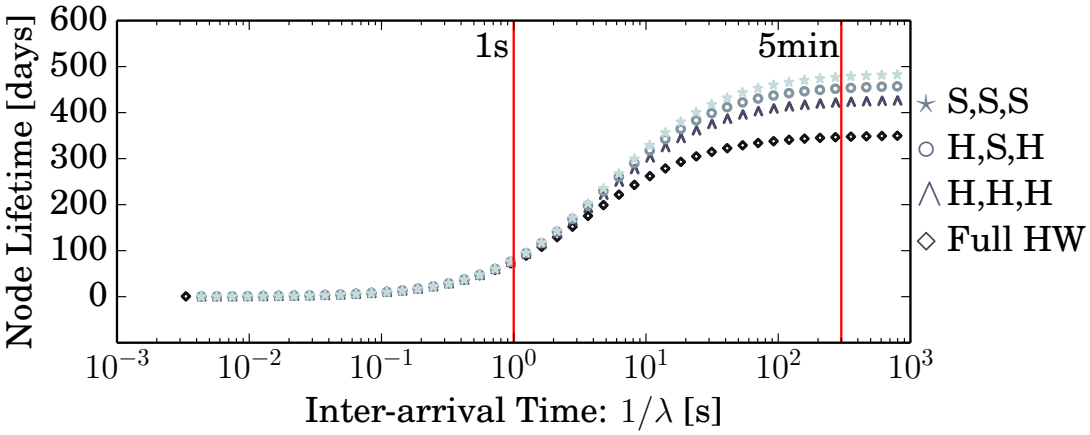
Our full-hardware implementation is roughly 180k transistors, effectively 30 % of the reported 592k transistors comprising ULSNAP [54]. We assume that the BS computation is parallelized and requires four S-box. The Ctrl unit is roughly double the size of the sum total of all individual blocks. Unsurprisingly, Table 6.8 shows an all-hardware implementation offers the best throughput. The key contributor to performance seems to be the hardware-based control. Moving to software-based control of the hardware results in a 10x penalty to throughput. However, a full-hardware solution also has the *lowest* mote lifetime as most the time the mote is idle and static power consumption dominates.

Fig. 6.6a shows the lifetime of various hybrid AES configurations alongside ULSNAP as function of the event inter-arrival time ($1/\lambda$). For legibility, we omit some of the hybrid configurations—they cluster together rather tightly. For $1/\lambda < 4\,\mathrm{s}$, the hybrid schemes offer better performance than a software-only implementation with little or no impact in the lifetime of the mote. In contrast, for sparse events t_{life} is governed by the static power. For inter-arrival times greater than 5 minutes, Fig. 6.6a shows a gap in excess of 3x between the hybrid and software counterparts.

One solution to reduce leakage current in the hardware portions of hybrid configurations is to use power gating. The simple addition of cut-off transistors in QDI circuits can reduce static power consumption by an average of 80 % with an average of 20 % performance degradation as explained in Chapter 4 and in [55]. Using these estimated power savings to evaluate the mote lifetime, we obtain the plots in Fig. 6.6b. The increased lifetime of the hybrid and all-hardware implementations suggests that our AES implementation can greatly benefit from power



(a) Lifetime



(b) Lifetime with Power Gating

Figure 6.6: ULSNAP Lifetime

gating, particularly when the mote stays idle for long periods of time.

Figures 6.6a and 6.6b assume that the software portions of the hybrid AES implementations share memory with the host microprocessor, in this case ULSNAP. If we require dedicated memory for the AES co-processor, we must account for the additional power consumption of this dedicated memory. We assume 120 pA of leakage current per bitcell [18]. This additional memory does not improve the throughput of the AES system, but it does free memory for other tasks on the processor. Fig. 6.7 illustrates the effect of incorporating the static power consumption

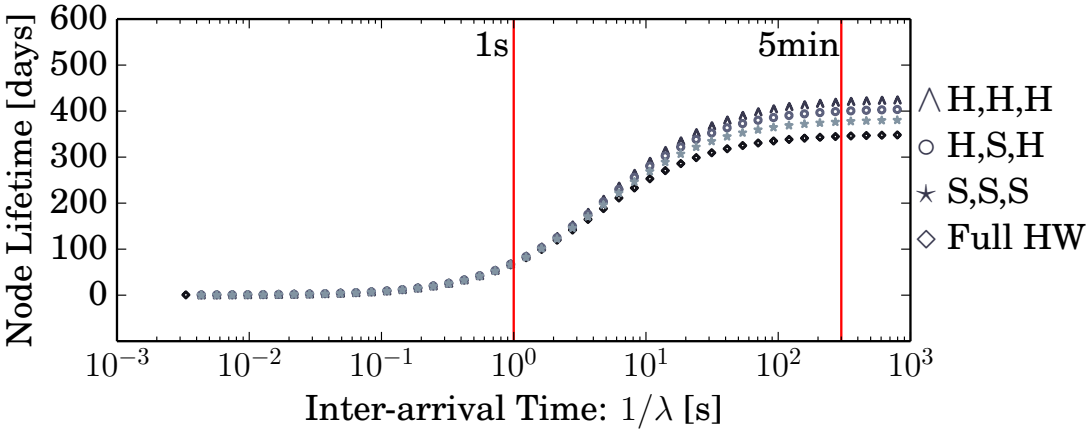


Figure 6.7: ULSNAP Lifetime with Memory Overheads

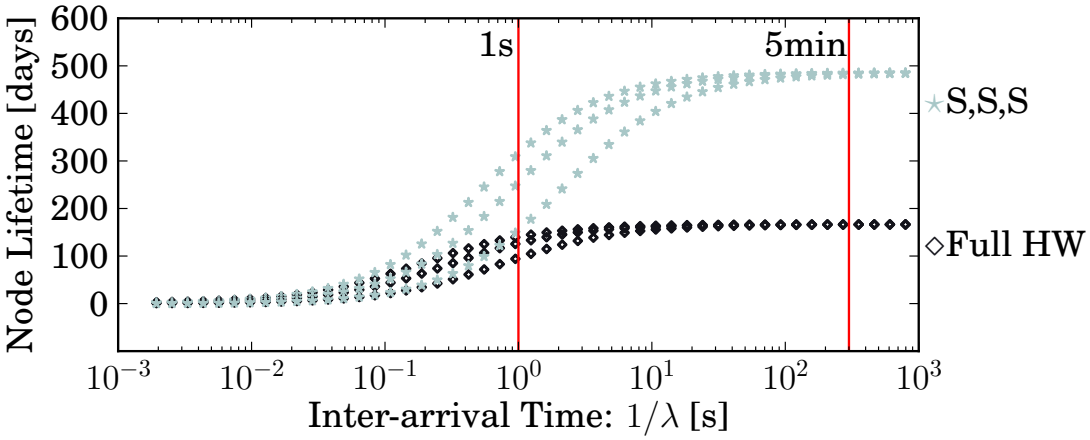


Figure 6.8: ULSNAP Lifetime for $\alpha \in \{0.01, 0.1, 0.4\}$

of the additional SRAM as listed in Table 6.7. In this regime, the hybrid modes actually offer the best lifetime as the high memory requirements of the software-only mode increase the static power to the point where it is only 10 % better than the all-hardware configuration.

Fig. 6.9 illustrates the effects of the probability of sending a packet, α, on mote lifetime for a full software and full hardware configuration of our AES implementation alongside ULSNAP. $\alpha = 1$ means we always send a packet after receiving a sensor event. Lower values of α represent less transmissions and thus longer mote

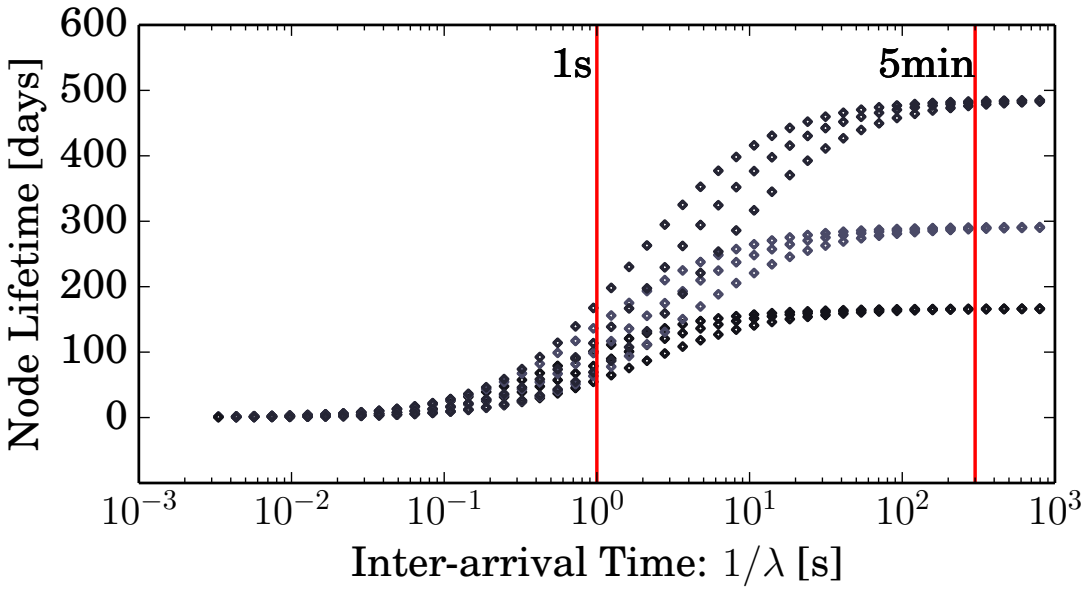


Figure 6.9: ULSNAP Lifetime for $\alpha \in \{0.01, 0.1, 0.4\}$

lifetime. What is interesting is that as the inter-arrival time of events increases the effect of α diminishes. This represents the increasing dominance of static power consumption for high values of $1/\lambda$. Not shown here is the effect of packet payload size, which looks quite similar to the plots in Fig. 6.9. Intuitively, one can approximate a larger packet with many smaller packets.

CHAPTER 7

CONCLUSION

'Begin at the beginning,' the King said gravely,
'and go on till you come to the end: then stop'

Lewis Carroll

This thesis presents the design of a state-of-the-art of a microcontroller that fits the computational paradigm found in Wireless Sensor Networks well. We exploit the ULSNAP architecture to design and implement a chip that provides high performance, low energy and improved node lifetime over available commodity processors. Our 90 nm test chip performs in the Pareto-optimal set of the energy-performance curve. ULSNAP's event-driven architecture matches the low power, bursty performance requirements of sensor network applications.

Microarchitectural choices such as splitting the operand buses into a hierarchical slow/fast bus design further reduced the energy and improved average case performance. We maximized the energy efficiency of our memory by banking each memory module and decoupling the buses, activating only accessed banks. Furthermore, we enabled multiple outstanding memory operations to be executed by implementing the memory access modules with full buffers. We further improved the idle power by minimizing the static power consumption of the memory banks. Additionally, ULSNAP's timer and I/O coprocessors were optimized to minimize the power in the quiescent state by, among other things, decoupling their control from the core, and decoupling control between the available timers.

At the circuit level, we leveraged the strengths of our self-timed circuits. Their self timed nature means that they are event-driven without any additional control

overheads. QDI circuits minimize power consumption during idle periods of time and automatically adjust to variations in voltage and other environmental factors.

This thesis presents *measured* results for ULSNAP: A fully implemented Ultra-Low power Event-Driven Sensor Network Asynchronous Processor offering high performance within a low energy envelope. With respect to the state-of-the-art processors in its class, ULSNAP offers Pareto-optimal operating points in the energy-performance space. It achieves 93MIPS at 1.2 V and 47MIPS at 0.95 V while using 47 pJ and 29 pJ per cycle respectively.

We adapted a WSN mote battery lifetime model that accounts for the energy required to process, encrypt, and transmit collected data as well as the energy consumption during idle periods. We use this model to evaluate the lifetime of a mote that uses state-of-the-art microcontrollers. Our results show that our test chip has a lifetime of 2.5 years when events arrive every 10 s and the time to process and event is 1 ms—a common workload pattern in the WSN application space.

Furthermore, we analyzed and designed a cryptographic system that is suitable for the WSN design space. Cryptography is a *critical* block within the WSN as encryption is arguably the first line of defense to protect the confidentiality of data transmitted over the wireless link.

If throughput is the main concern, our full hardware implementation of AES delivers 30× *net* performance over its software counterpart. On the other hand, a complete hardware implementation is unattractive from the standpoint of mote lifetime due to the increase in leakage current from the additional transistors.

If encryption throughput is not high-priority, a complete software AES implementation can offer a 3× increase in mote battery lifetime. Incorporating power

gating techniques into our hybrid and full hardware designs significantly reduced the gap in battery lifetime to less than 66 %. If necessary, we can provide dedicated memory resources to the AES implementation but at the cost of increasing leakage current to the point where a fully software implementation is only 10 % better than a hardware implementation. A hybrid hardware/software implementation gives 6× net performance improvement, and increases lifetime by 10 % over the full software counterpart.

In this thesis, we advance the state of the art by providing the design and implementation details of an energy efficient, high performance processor that fits the wireless sensor node design space. The designs and experiments shown in this picture will enable a smart, instrumented, and connected world that will weave smart elements into the fabric of everyday life.

APPENDIX A

COMMUNICATING HARDWARE PROCESS

The CHP notation we use is based on Hoare's CSP [19]. A complete formal semantics of the language can be faound in van der Goot's What follows is a short and informal description.

Simple Statements

- Skip: *skip*. This statement does nothing.
- Assignment: $a := E$. This statement means "assign the value of expression E to a. $a\uparrow$ is shorthand for $a := true$, and $a\downarrow$ for $a := false$.

Control Statements

- Selection: $[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$. Where G_i are boolean expressions (guards) and S_i are program parts. Execution stalls until a Gi is true, at which point Si is executed. The notation $[G]$ is short-hand for $[G \rightarrow$ skip$]$, which stalls until $G = true$. If guards are not mutually exclusive, we use the vertical bar $\vee$ instead of $[]$. The selection statement is assumed to be demonic, and it is therefore not fair.
- Repetition: $*[G1 \rightarrow S1 \; [] \; ... \; [] \; Gn \rightarrow Sn]$. Choose $Gi = true$, execute Si. Repeat until no Gi is true. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$. If the guards are not mutually exclusive, the use the vertical bar $\vee$ instead of $[]$.
- Send: $X!E$. Evalute expression E and send result over channel X. Both, send and receive are blocking, enabling them to be used as both synchronization and data-communication primitives.

- Receive: $Y?v$. Receive value over channel Y and store variable v.
- Probe: $\overline{X}$ is a boolean which is *true* if and only if a communication over channel X can complete without suspending. Probes are only allowed to occur in the guards of choice statements.

Statement Composition

- Sequential Composition: $S; T$. The execution of this command corresponds to executing S followed by T. The semicolon binds tighter than the parallel composition operator "‖", but weaker than the comma or bullet.
- Parallel Composition: $S \parallel T$ or S, T. The execution of this command corresponds to executing commands S and T in parallalel. The "‖" operator binds weaker than bullet or semicolon. The comma binds tighter than the semicolon but weaker than the bullet. The parallel execution of CHP process is assumed to be weakly fair – every enabled action will be given a change to execute eventually.
- Simultaneous Composition: $S \bullet T$ both S and T. This command corresponds to the execution of S and T actions that complete simultaneously. Typically S and T are both communication actions.

APPENDIX B

INSTRUCTION SET ARCHITECTURE

B.1 State of the Processor

ULSNAP instructions can be one or two words long. The processor has a set of 16-bit general purpose registers, `reg[0] ... reg[16]`.

The instructions are stored in a 4 kB, single-cycle SRAM. Data segments and memory variables are stored in a 4 kB data SRAM. Each word in memory is 16 bits. All accesses to SRAM are word aligned.

An internal register holds the value of the current program counter `PC` and a status register `carry` holds the output carry of the last arithmetic operation.

B.2 Instruction encodings

ULSNAP implements a 16-bit *Load/Store* architecture and has a MIPS-like RISC ISA. All values must be present in the register file before performing an operation and results are transfered to and into memory using `LOAD` and `STORE` instructions. The list of instruction encodings is shown in Table B.1 and Table B.2. There are some changes with respect to the original SNAP ISA defined in [30]. The ULSNAP ISA doesn't implement complicated timer operations (`TCSHIFT`, `TCREAD`), nor allows to read the status of the the output queue (`READ`). On the other hand the ULSNAP ISA has an external instruction to control the external data memory register `LDR`.

Table B.1: Instruction encodings for single word instructions

Instruction	15..12 A	11..8 B	7..4 C	3..0 D
Register-Register ALU operations				
ADD	0000	dst	src1	src2
SUB	0001	dst	src1	src2
ADDC	0010	dst	src1	src2
SUBC	0011	dst	src1	src2
OR	0100	dst	src1	src2
AND	0101	dst	src1	src2
XOR	0110	dst	src1	src2
NOR	0111	dst	src1	src2
SRLV	1000	dst	src1	samt
SRAV	1001	dst	src1	samt
SLLV	1010	dst	src1	samt
SRLI	1100	dst	src1	imm
SRAI	1101	dst	src1	imm
SLLI	1110	dst	src1	imm
Register-Register Timer and Interrupt operations				
RAND	1111	dst	—	0010
SCHEDHI	1111	dst	src	0100
SCHEDLO	1111	dst	src	0101
LDR	1111	—	src	0110
CANCEL	1111	id	0000	0111
SEED	1111	src	0001	0111

B.3 Calling Conventions

ULSNAP uses LCC compiler toolchain to compile a C program into ULSNAP's assembly code. A custom-built assembler builds the object and image files that can be loaded into ULSNAP on reset. Our assembler and linker automatically performs some static and peephole optimizations.

The ULSNAP programming specification enforces the calling conventions in Table B.3. All C or assembly programs should follow these to be compatible with current and future ULSNAP libraries.

Table B.2: Instruction encodings for double word instructions

Instruction	15..12 A	11..8 B	7..4 C	3..0 D
Double word instructions				
ADDI	1011	dst	src1	0000
ADDIC	1011	dst	src1	0010
ORI	1011	dst	src1	0100
ANDI	1011	dst	src1	0101
XORI	1011	dst	src1	0110
NORI	1011	dst	src1	0111
LOADD	1011	addr	dst	1100
STORED	1011	addr	src	1101
LOADI	1011	addr	dst	1110
STOREI	1011	addr	src	1111
BFS	1011	st	src1	1010
SETADDR	1011	—	src1	1000
Control Instructions				
BEQ	1111	src2	src1	1000
BNE	1111	src2	src1	1001
BGEZ	1111	src1	—	1010
BLTZ	1111	src1	—	1011
JAL	1111	dst	—	1100
JALR	1111	dst	src	1101
WAIT	1111	—	—	1110

Table B.3: ULSNAP calling conventions

Register	Convention
R0	Always 0
R1	Bulk copy register
R2	Return value, caller saved
R3-R5	Function arguments
R6-R8	Temporaries, caller saved
R9-R12	Register, callee saved
R13	Stack pointer
14	Return address
15	I/O Register

APPENDIX C

ULSNAP'S DEVELOPMENT BOARD

The ULSNAP development board has the form factor of an Arduino Mega 2560 shield. We use the USB capabilities of the Arduino to connect the ULSNAP shield to the computer via a virtual serial port.

To measure the power consumed by ULSNAP, the core power supply is connected in series with a $1\,\Omega$ resistor. The voltage drop is amplified by the MAX208 instrument amplifier. The ADC on the Arduino board is able to measure the voltage amplified drop across the $1\,\Omega$ resistor. We performed a fine-calibration of the ADC by empirical tests.

Table C.1 shows the full Bill of Materials (BOM) that we used to build the board. Fig. C.1 shows the breakout connection between the ULSNAP board and the Arduino board. Fig. C.2 shows the power regulators, and adapter that allows us to use the Arduino power in the ULSNAP board. Fig. C.3 shows the schematic connections between ULSNAP, the TI TXB0108 voltage-level translators and the Arduino board.

The finalized layout of the PCB is shown in Fig. C.5. The populated board is two layer board with solder mask and silk screen. The dimmensions of the board 10.2 cm on one side and 5.5 cm on the other. The PCB has breakout headers to be able to stack extra Arduino shields on top of it. Stacking extra shields, allowed us to easily connect an LCD as well as multiple sensors and actuators.

Part	Description	Value
Capacitor	Tantalum Capacitor for Voltage Regulator Output	4.7 µF
Capacitor	Power supply's bypass capacitor	4.7 µF
Capacitor	Temperature sensor's bypass capacitor	4.7 µF
Header 1x2	Right angle header for source meter unit	
Header 1x3	Right angle header for custom delay chain	
Header 1x4	Right angle header for custom chip select	
Header 1x5	Right angle header for custom chip select	
LED	Power-On LED indicator	3.92 mcd
MAX 6627	Temperature Sensor	
Push Button	Reset Button	
Resistor	Power-On LED resistor	1.6 MΩ
Resistor	Power source measurement	1 Ω
Switch	Power source select	
Socket 48pin	48 pin DIP socket for ULSNAP test chip	
TI TPS77012	Power Regulator	
TI TXB0108	Bi-directional voltage-level translators	
MAX4208AUA	Instrument amplifier to measure voltage	
ULSNAP test chip		

Table C.1: BOM for ULSNAP Arduino Shield test board

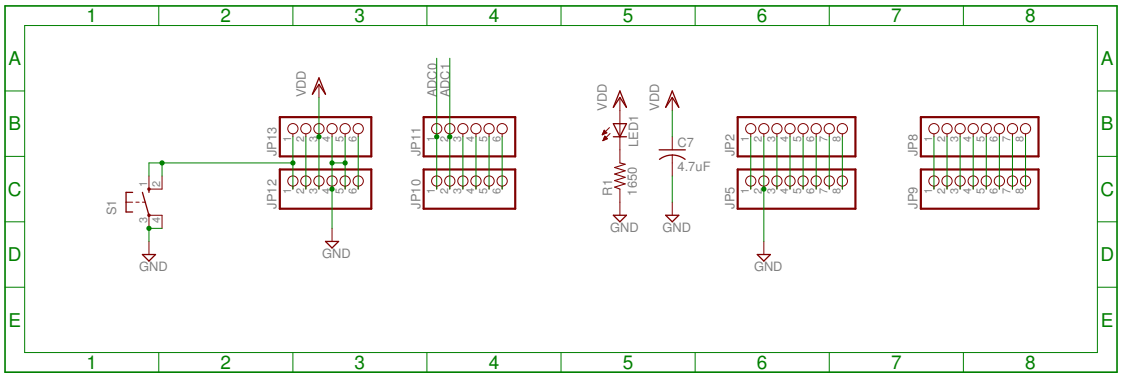


Figure C.1: PCB connections to Arduino Mega 2560 board

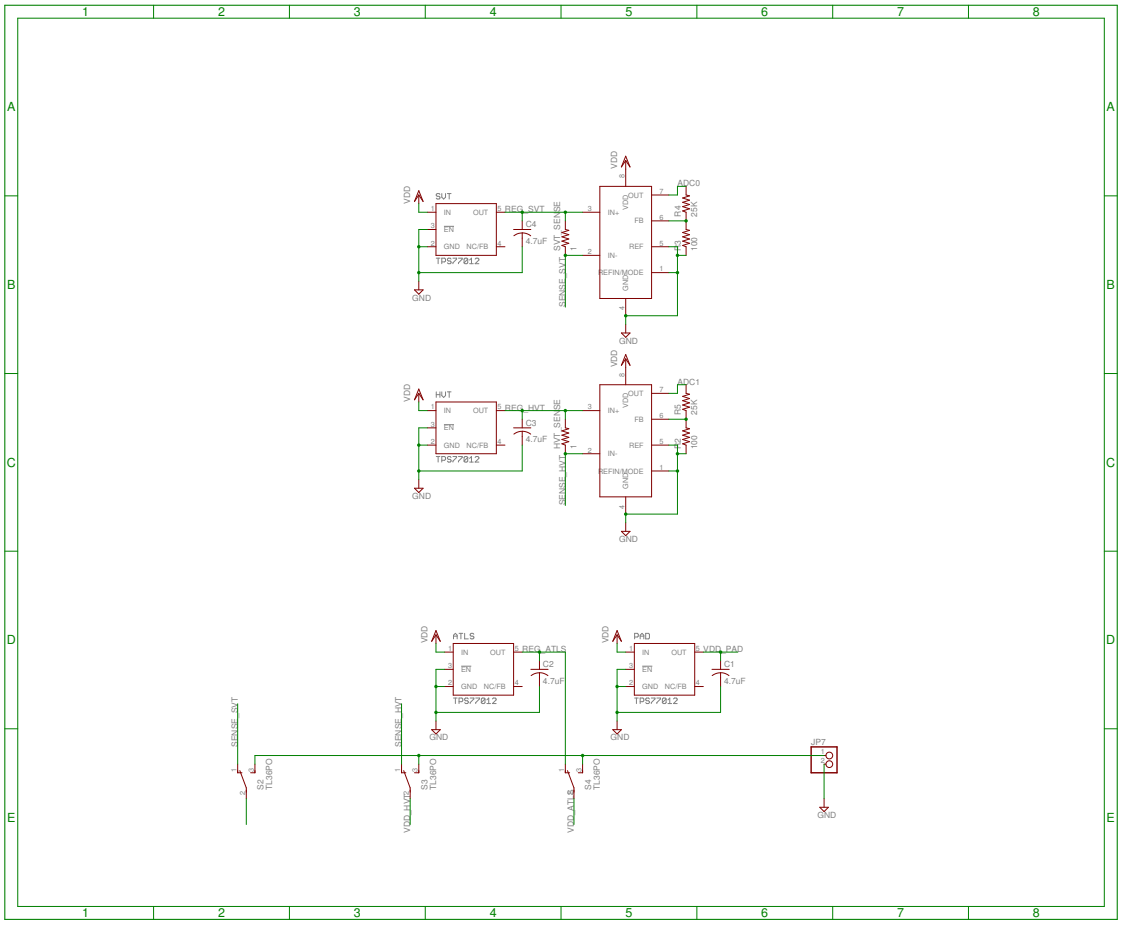


Figure C.2: Power regulators and power control switches

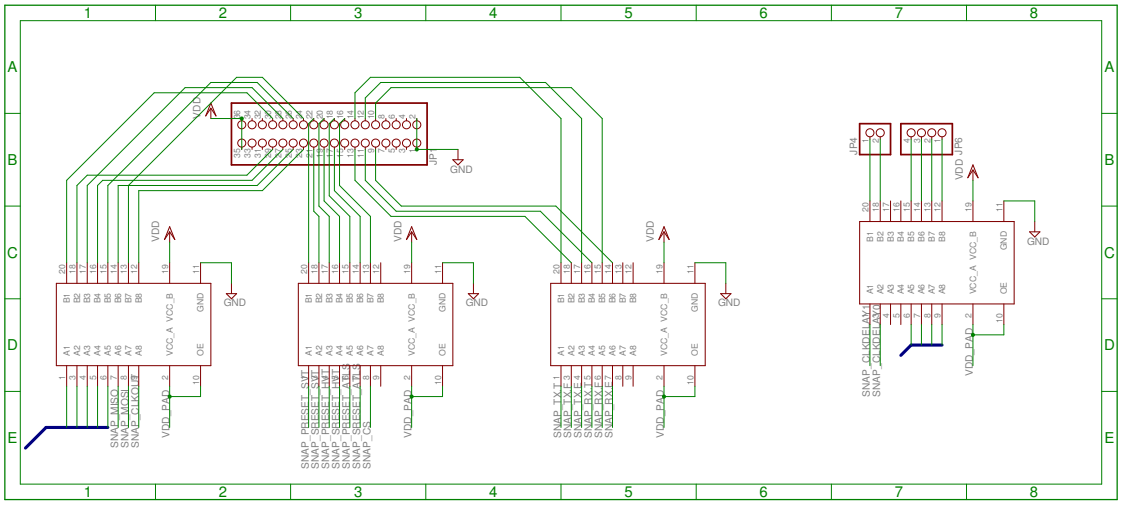


Figure C.3: Bidirectional voltage-level translators

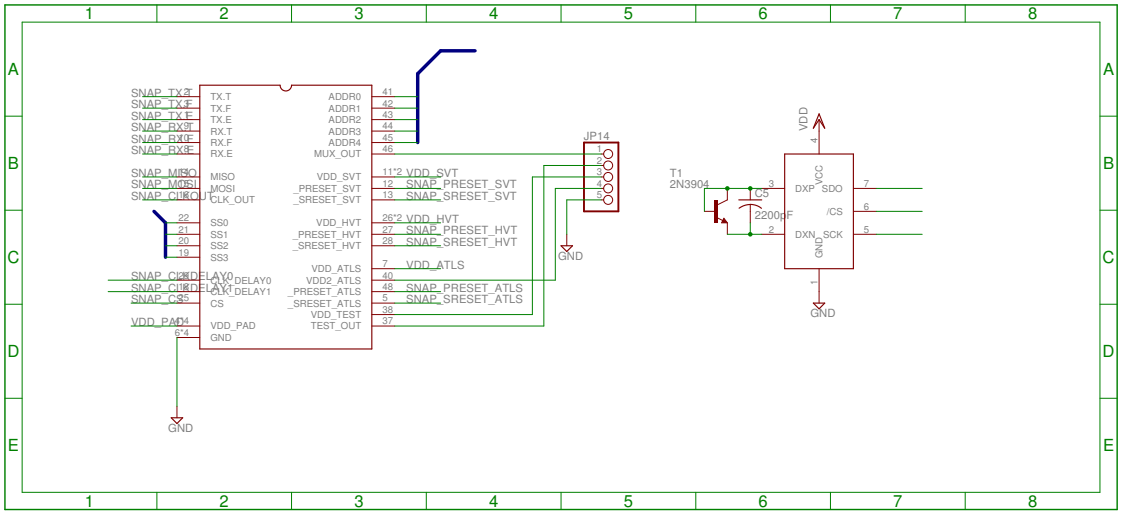


Figure C.4: ULSNAP-Arduino connection

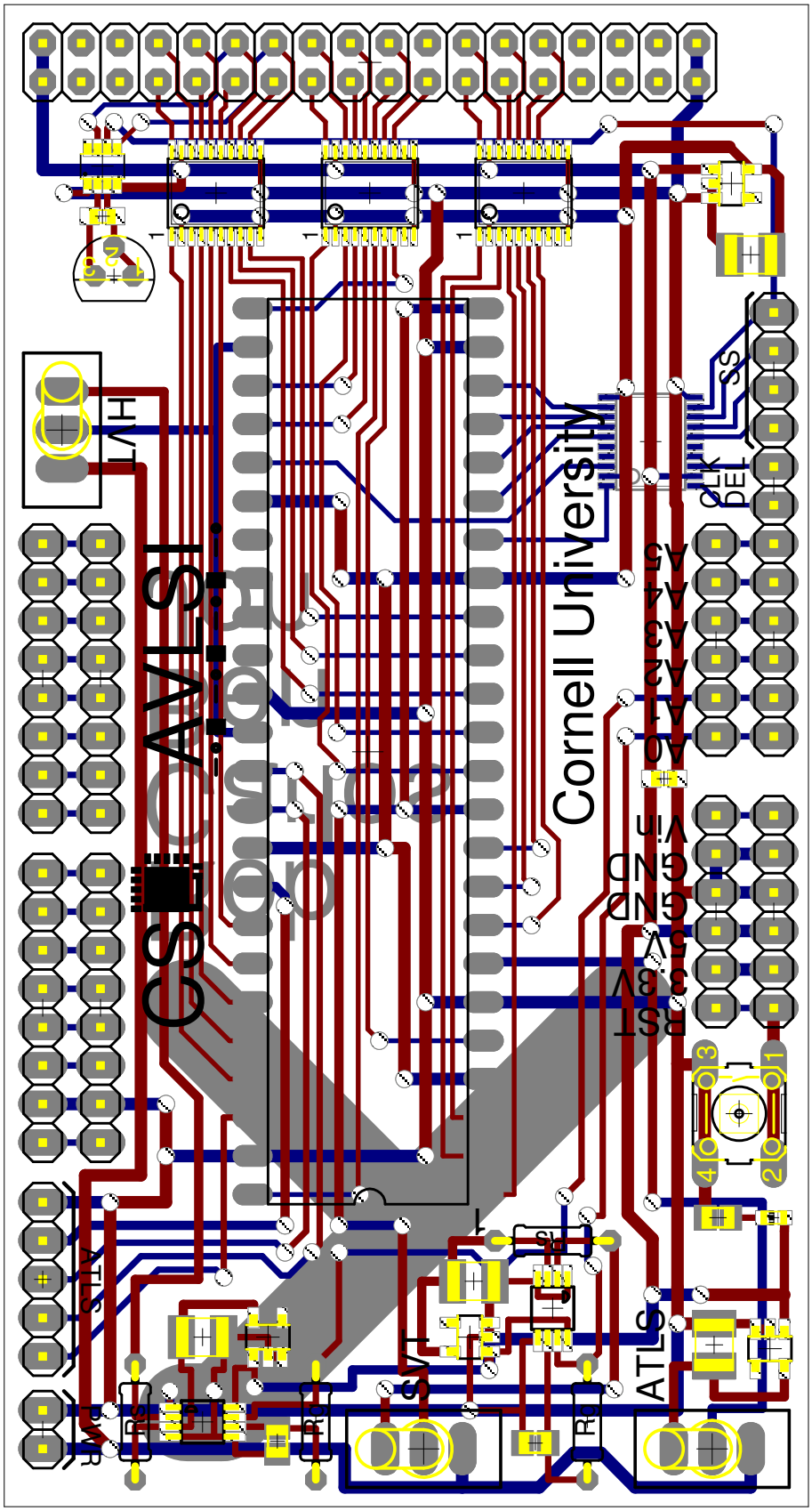
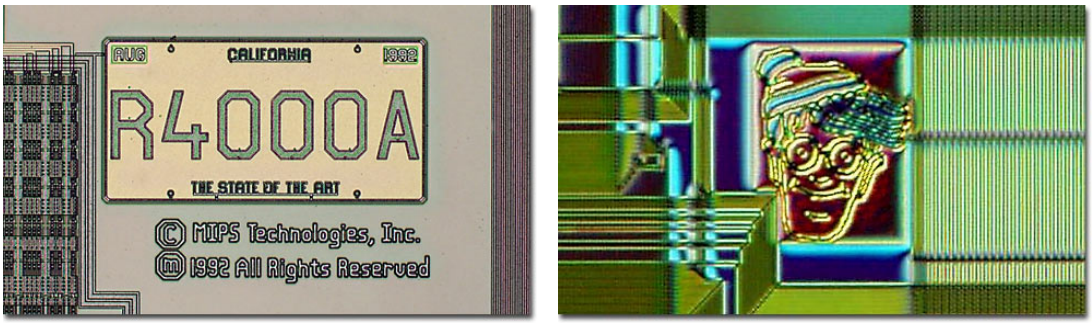


Figure C.5: Layout of ULSNAP's test chip PCB board

APPENDIX D

CHIP ART

Chip Art refers to the nanometric "art work" drawn into Integrated Circuits [10]. In the past, chip art thwarted illegal IP infringement by competitors [16]. This is not the case since the Chip Protection Act of 1984. Nowadays, most chip art exists as a tribute to the design teams that want to leave a mark for their own. These doodles and letters give a sense of pride to the designers that a signature gives to an art painter. Notable examples of chip art include the artwork in the MIPS R4000 chip and "Where is Waldo", shown in Fig. D.1.



(a) MIPS4000 chip art resembling a license plate

(b) We found Waldo in a Silicon Graphics chip

Figure D.1: Samples of chip art commercially available microcontrollers

The stricter DRC-rules, the heavy use of EDA tools, shorter design time cycles, large engineering teams, and a fierce competition are slowly erasing this esoteric and sophisticated practice. Figs. D.2, D.3, are a few samples of chip-art from the prototypes manufactured in this work.

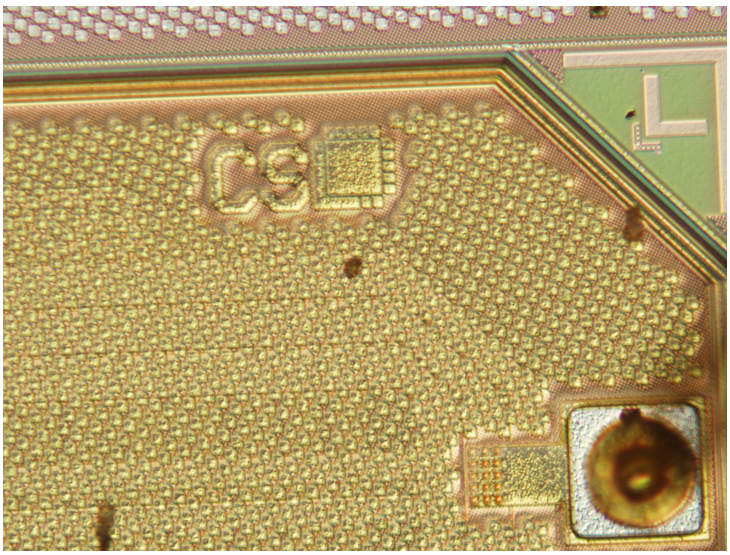


Figure D.2: Computer Systems Laboratory 2010 logo

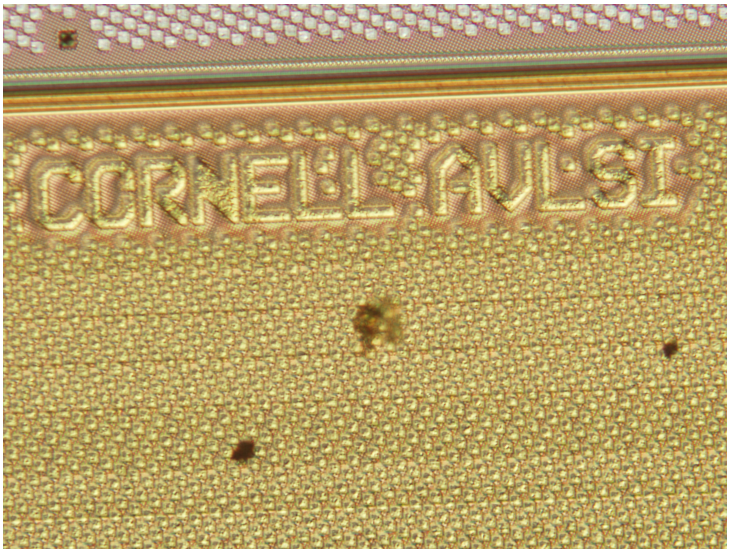


Figure D.3: AVLSI Group logo

BIBLIOGRAPHY

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 2002.

[2] M. Ashouei, J. Hulzink, J. Van Ginderdeuren, and et al. A voltage-scalable biomedical signal processor running ECG using 13pJ/cycle at 1Mhz and 0.4V". *International Solid-State Circuits Conference*, 2011.

[3] Atmel. ATSAM— ARM-based Flash MCU, SAM4L.

[4] Atmel. ATtiny13A Complete Datasheet.

[5] Atmel. AVR4013: picoPower Basics.

[6] E. Biham, A. Biryukov, and A. Shamir. Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials - Springer. *Journal of Cryptology*, 2005.

[7] M. Bishop. *Introduction to Computer Security*. Addison-Wesley Professional, 2004.

[8] A. Bogdanov, D. Khovratovich, and C. Rechberger. Biclique cryptanalysis of the full aes. In *Advances in Cryptology — ASIACRYPT 2011*, pages 344–371. Springer, Berlin, Heidelberg, 2011.

[9] G.F. Bouesse, M. Renaudin, A. Witon, and F. Germain. A clock-less low-voltage aes crypto-processor. In *ESSCIRC*, 2005.

[10] M. Byko. Designers make their mark with computer chips. *Journal of the Minerals, Metals, and Materials Society, JOM*, 2001.

[11] G. Chen, M. Fojtik, D. Blaauw, and et al. Millimeter-scale nearly perpetual sensor system with stacked battery and solar cells. *International Solid-State Circuits Conference*, 2010.

[12] V. Ekanayake, C. Kelly, and R. Manohar. An ultra low-power processor for sensor networks. *ASPLOS: Architectural Support for Programming Languages and Operating Systems*, 2004.

[13] S.Z. Fatemian and D. Hatzinakos. A new ecg feature extractor for biometric recognition. In *16th International Conference on Digital Signal Processing*, July 2009.

[14] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. Aes implementation on a grain of sand. *Information Security, IEEE Proceedings*, Oct 2005.

[15] A. M. Feldman and R. Serrano. Welfare economics and social choice theory. 1980.

[16] H. Goldstein. The Secret Art of Chip Graffiti. *IEEE Spectrum*, 2002.

[17] S. Hauck. Asynchronous design methodologies: an overview. *Proceedings of the IEEE*, 1995.

[18] D. Ho, K. Iniewski, S. Kasnavi, A. Ivanov, and S. Natarajan. Ultra-low power 90nm 6T SRAM cell for wireless sensor network applications. In *ISCAS*, 2006.

[19] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 1978.

[20] L.A Hollaar. Direct implementation of asynchronous control units. *IEEE Transactions on Computers*, Dec 1982.

[21] Masashi I., Kouei T., and Takashi N. Fine-grain leakage power reduction method for m-out-of-n encoded circuits using multi-threshold-voltage transistors. *ASYNC: IEEE International Symposium on Asynchronous Circuits and Systems*, 2009.

[22] N. Ickes, Y. Sinagil, F. Pappalardo, E. Guidetti, and A. Chandrakasan. A 10 pJ/cycle ultra-low-voltage 32-bit microprocessor system-on-chip. *European Solid-State Device Conference*, 2011.

[23] Crossbow Technology Inc. Wireless Measurement System: MICA2.

[24] Texas Instruments. CC1101 - Low-Power Sub-1 GHz RF Transceiver.

[25] Texas Instruments. CC430F6137 - MSP430 SoC With RF Core.

[26] Texas Instruments. MSP430L092 Mixed Signal Microcontroller.

[27] D. Jung, T. Teixeira, and A. Savvides. Sensor node lifetime analysis: Models and tools. *ACM TOSN*, 2009.

[28] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *SenSys*. ACM, 2004.

[29] Sean Keller, Siddharth Bhargav, Chris Moore, and Alain J. Martin. Reliable minimum energy cmos circuit design. *European Workshop on Variability (VARI)*.

[30] IV Kelly, C., V. Ekanayake, and R. Manohar. SNAP: a Sensor-Network Asynchronous Processor. *ASYNC: IEEE International Symposium on Asynchronous Circuits and Systems*, 2003.

[31] N. Kimura and S. Latifi. A survey on data compression in wireless sensor networks. In *International Conference on Information Technology Coding and Computing, ITCC.*, April 2005.

[32] Y.W. Law, J. Doumen, and P. Hartel. Survey and Benchmark of Block Ciphers for Wireless Sensor Networks. *ACM TOSN*, 2006.

[33] A. Lines. Pipelined asynchronous circuits, 1995.

[34] H. Lipmaa, D. Wagner, and P. Rogaway. Comments to NIST concerning AES modes of operation: CTR-mode encryption. Technical report, 2000.

[35] M. Luk, G. Mezzour, A. Perrig, and V. Gligor. MiniSec: A Secure Sensor Network Communication Architecture. In *IPSN*, 2007.

[36] R. Manohar. Pipelined Mutual Exclussion and the Design of an Asynchronous Microprocessor. *Computer Systems Laboratory, Technical Report CSL-TR-2001-1017*.

[37] R. Manohar, T.K. Lee, and A. Martin. Projection: A synthesis technique for concurrent systems. In *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999.

[38] A. Martin, A. Lines, and R. Manohar. The design of an asynchronous MIPS R3000 microprocessor. *Advanced Research in VLSI*, 1997.

[39] A. J. Martin. Compiling Communicating Processes for Delay-Insensitive VLSI Circuits. *Distributed Computing*, 1986.

[40] A. J. Martin. Developments in concurrency and communication. chapter Programming in VLSI: From Communicating Processes to Delay-insensitive Circuits, pages 1–64. Addison-Wesley Longman Publishing Co., Inc., 1990.

[41] A. J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The Design of an Asynchronous Microprocessor. *SIGARCH Computer Architecture News*, 1989.

[42] AJ. Martin, M. Nystrom, and C.G. Wong. Three generations of asynchronous microprocessors. *Design Test of Computers, IEEE*, Nov 2003.

[43] J.D. Meindl and J.A Davis. The fundamental limit on binary switching energy for terascale integration (tsi). *Solid-State Circuits, IEEE Journal of*, 35(10):1515–1516, Oct 2000.

[44] Microchip. PIC10(L)F320/322.

[45] S. Mingoo, S. Hanson, L. Yu-Shiang, F. Zhiyoong, K. Daeyeon, L. Yoonmyung, N. Liu, D. Sylvester, and D. Blaauw. The Phoenix Processor: A 30pW platform for sensor applications. In *VLSI Circuits, 2008 IEEE Symposium on*, 2008.

[46] S. Morioka and A Satoh. A 10 gbps full-aes crypto design with a twisted-bdd s-box architecture. In *IEEE International Coference on VLSI in Computers and Processors*, pages 98–103, 2002.

[47] L. Nazhandali, M. Minuth, and T. Austin. SenseBench: toward an accurate evaluation of sensor network processors. *IEEE IISWC*, 2005.

[48] NIST. *FIPS 46-3: Data Encryption Standard (DES).*

[49] NIST. SKIPJACK and KEA Algorithm Specifications Version 2.0. Technical report, May 1998.

[50] NIST. FIPS 197: Advanced Encryption Standard. 2001.

[51] NIST. SP 800-38A. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. 2001.

[52] NIST. SP 800-131A. Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. 2011.

[53] I. Onat and A. Miri. An intrusion detection system for wireless sensor networks. *Wireless And Mobile Computing*, 2005.

[54] C.T. Ortega Otero, J. Tse, R. Karmazin, B. Hill, and R. Manohar. ULSNAP: An Ultra-low Power Event-Driven Microcontroller for Sensor Network Nodes. *IEEE ISQED*, 2014.

[55] C.T. Ortega Otero, J. Tse, and R. Manohar. Static Power Reduction Techniques for Asynchronous Circuits. In *IEEE ASYNC*, 2010.

[56] D.A. Osvik, A. Shamir, and E. Tromer. *Cache Attacks and Countermeasures: The Case of AES*. Topics in Cryptology – CT-RSA2006. Springer Berlin Heidelberg, 2006.

[57] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J.D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Wireless Networks*, 2001.

[58] K.S.J. Pister, J.M. Kahn, B.E. Boser, et al. Smart dust: Wireless networks of millimeter-scale sensor nodes. *Electronics Research Laboratory Research Summary*, 1999.

[59] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 2000.

[60] Y.K. Ramadass and A.P. Chandrakasan. Voltage Scalable Switched Capacitor DC-DC Converter for Ultra-Low-Power On-Chip Applications. In *Power Electronics Specialists Conference, 2007. IEEE*, 2007.

[61] Renesas. RL78/G12 Datasheet: R01DS0207EJ0200.

[62] S. Romanovsky, A. Achyuthan, S. Natarajan, and Wing Leung. Leakage reduction techniques in a 0.13 um sram cell. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, 2004.

[63] I. Romero, T. Berset, D. Buxi, L. Brown, J. Penders, S. Kim, Nick Van H., H. Kim, C. Van Hoof, and F. Yazicioglu. Motion artifact reduction in ambulatory ecg monitoring: An integrated system approach. In *Proceedings of the 2nd Conference on Wireless Health*, New York, NY, USA. ACM.

[64] D. Saha, D. Mukhopadhyay, and D. R. Chowdhury. A Diagonal Fault Attack on the Advanced Encryption Standard. *IACR Cryptology ePrint*, 2009.

[65] M. Seok, S. Hanson, Y.S. Lin, et al. The Phoenix Processor: A 30pW platform for sensor applications. In *IEEE ISVLSI*, 2008.

[66] Ho-Jun Song. A self-off-time detector for reducing standby current of DRAM. *IEEE SSC*, 1997.

[67] I. Verbauwhede, P. Schaumont, and H. Kuo. Design and Performance Testing of a 2.29-GB/s Rijndael Processor. *IEEE JSCC*, 38:569–572, 2003.

[68] T. Verhoeff. Delay-insensitive codes an overview. *Distributed Computing*, 1988.

[69] B.A. Warneke and K.S.J. Pister. An ultra-low energy microcontroller for Smart Dust wireless sensor networks. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, 2004.

[70] M. Weiser. The computer for the 21st century. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.

[71] M. Wieckowski, G.K. Chen, S. Mingoo, D. Blaauw, and D Sylvester. A hybrid DC-DC converter for sub-microwatt sub-1V implantable applications. In *Symposium on VLSI Circuits*, 2009.